An Introduction to PlayStation®2 Programming

(Part 1)

James Russell
SCEE Technology Group
`James_Russell@scee.net`

V0.94 – December 2000

About this document

This tutorial series is aimed at programmers new to PlayStation 2 console development. It is split into 2 parts.

Part 1 – [This document] This is aimed at new PlayStation 2 programmers. It assumes the reader knows about most of the concepts involved with console games development, like double buffering graphics and C compiling. It is beneficial if the reader has had some experience with assembly language.

This document is archived with 7 example programs. So that the reader can examine the examples at their leisure rather than be tied to a development kit, the full source to each example is included in this document.

Part 2 – Development strategies. This is aimed at developers who are confident with the system components. It covers ways of designing PlayStation 2 applications to achieve high performance.

Document organisation

This document is split into various Sections.

Section A is concerned with the development kit setup.

Section B contains an overview of the architecture.

Section C is about the tool chain and how to use it.

The document then attempts to explain the various components of the system, and how they are used to build programs. Examples are used which perform simple tasks (like putting a polygon on the screen), and the concepts in each example are built upon in later examples.

The source code for each example is contained in Appendix I.

Section D - Example 1 puts 2 polygons on the screen. It introduces the concepts of GIF tags, REGLIST mode and DMA lists.

Section E - Example 2 performs the same task. It introduces DMA tags and the GIF's PACKED mode.

Section F - Example 3 rotates a 3D square on the screen. It introduces the VU0 macromode library, and various VU0 concepts.

Section G - Example 4 uses a VU0 micromode program to perform the 3D calculations. It introduces VU programming and the `ee-dvp-as` assembler.

Section H - Example 5 introduces the VIFs, and uses VIF0 and the DMA to upload and activate a VU0 microprogram.

Section I - Example 6 performs the same task, except it uses VU1 to perform the calculations and create the GS packets. It introduces PATH1 and the `XGKICK` instruction.

Section J - Example 7 performs the same task as example 6, except that is uses half the VRAM using Frame based drawing instead of Field based drawing.

Section K wraps up some loose ends regarding PS2 development.

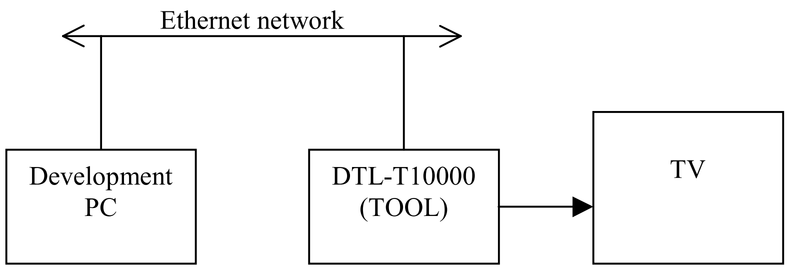
Section L concludes the document, and discusses what other areas of development could be explored next.

Appendix I contains full listings of the source code to all examples.

Section A – Setting up

You will need the following to program the PlayStation 2.

- A DTL-T10000 development kit (commonly referred to as a TOOL, TOOL, T10000 or T10K). We'll refer to it as a TOOL. This is the big black box that looks like an upright PlayStation 2 on steroids. The TOOL will be connected to a TV so you can see the output, and to an Ethernet network so that you can upload programs to it. The TOOL uses TCP/IP to communicate with other computers on the network. You do not work at the TOOL in the same way you would work at a PC. Instead, you upload programs to it over the network from a development PC.
- A development PC. You write and compile your programs on your development PC, and upload it over the network to the TOOL. Your development PC can run Linux or Windows, but the compilers that are supplied by SCE are all Linux based. 3rd party solutions such as SN Systems' "Pro-DG" and Metrowerks' "Codewarrior" run in a Windows environment. Almost any version of Linux will do. Developer Support use RedHat 6.2. This can be obtained from `www.redhat.com`.
- Development tools. SCE supplies the standard toolset, which run on Linux. For the examples in this document, we'll assume you're using the Linux tools. The latest versions can be obtained from the support website (`www.ps2-pro.com` for Europe/Oceania and `www.devnet.scea.com` for the USA).



Support websites and newsgroups

Each region of SCE has a support website, and SCEA/SCEE have unified private newsgroups.

The European Developer Support site is located at `https://www.ps2-pro.com/` and the newsgroups are located at `snews://news.ps2-pro.com` . Both of these sites are SSL secure, so you will need a SSL capable browser to access them. To obtain a company-wide username/password, please email `webmaster@scee.sony.co.uk`.

The US Developer Support site is located at `http://www.devnet.scea.com` and the newsgroups are at `snews://news.devnet.scea.com`. Contact `ps2_website@playstation.sony.com` for access information.

Preparing the TOOL

Connecting to a TV

Connect a television to the development kit via the AV Multi Out socket. This is the same socket as used on the back of a PlayStation or PlayStation 2. It can connect the TOOL to a variety of outputs, including composite, S-Video, component and RGB.

If possible, use RGB as it will give you the best possible signal. If your television does not have RGB input (many US and Japanese TVs do not), then use Component video or S-Video, as these are also high quality. However, it can be useful to test graphic designs using composite and/or RF modulation, as this is the format most consumers use to connect the PlayStation 2 to their televisions.

By default, the TOOL outputs an NTSC signal. Most modern TVs in Europe can handle PAL and NTSC, so if you are connecting by RGB or S-Video, the TV will display the colours and refresh rate correctly. If you are connecting by other means, the picture will be in black and white for NTSC output. You can switch the default output to PAL by changing a dip switch on the back of the TOOL. Consult the TOOL manual for details.

Connecting to the network

Connect your TOOL to the network via the Ethernet port on the back of the machine, and turn on the main power supply switch at the back. Then press and hold the power button on the top left of the TOOL for a second or two, until the machine beeps and the green light turns on. The TOOL is booting.

By default, the TOOL will attempt to obtain an IP address via DHCP (Dynamic Host Configuration Protocol). This may take some time. If it does not find a DHCP server or does not successfully negotiate an IP address, then its IP address will be set to the default (192.168.0.10). When the TOOL has completed booting, the TV screen will display its current IP address.

Configuring the TOOL

The TOOL runs a webserver that manages a web-based interface for configuring the TOOL and upgrading the software inside it. Enter the IP address of the TOOL into the location field of a web browser. For example, if your TOOL displays 192.168.0.10 as the IP address, enter "http://192.168.0.10/" into the location field. If you wish the TOOL to have a static IP address, you can use the web interface to set this address.

If you cannot contact your TOOL over the network, this may be because it is not on the same subnet as the rest of your network. Temporarily change your PC's IP address to one that is on the same subnet (e.g. if the TOOL displays an IP address of 192.168.0.10, change your PC's IP address to 192.168.0.1).

Complete details about TOOL setup are covered in the manual that comes with it. The rest of the document assumes that the TOOL has been assigned an IP address that your development PC can connect to. It is helpful for your system administrator to assign the TOOL a DNS hostname, so you do not have to remember the TOOL's IP address.

Inside the TOOL

The TOOL actually contains two computers. One is a PlayStation 2 with 128MB of EE RAM and 8MB of IOP RAM. It runs a slightly different operating system to the consumer PS2, and its ROMs are re-flashable.

The second computer is a PC. This is running Linux, and is connected to an Ethernet card and the IOP. It runs a server called 'dsnetm' that receives and executes instructions via a proprietary protocol called `deci2`. This protocol is used for transferring programs and data between the IOP and your computer, and also includes commands to start and debug programs.

The IOP on the TOOL is running an operating system that supports the `deci2` protocol, and the Linux tools all use `deci2` to communicate with the TOOL. This allows you to debug programs from your development PC.

Obtaining the development tools

There are two parts to the standard development software:

- The development tools (compilers, assemblers, debuggers)
- The development libraries (for writing PS2 programs)

These are distributed in two separate archives. This is because the libraries are updated more often than the tools. The tools are distributed in an archive called the Tool Chain, and the libraries are distributed in an archive called the Tool Libraries.

Documentation is sometimes contained in other archives. This is because Developer Support puts the Japanese versions of the tools and libraries on the website as soon as they arrive, and the English translations follow as soon as they are available.

If you have access to the support website (see above for the URL), go to the downloads section and enter the `Latest_Libs_and_Docs` directory (SCEA website users may have to access a different area). This directory contains all the archives and documents you will need to get a basic development system running on Linux.

In this directory there are many files with a prefix like **tc_160**. This would be part of the **T**ool **C**hain, version 1.60. The other prefix may look like **tlib_200**. This is part of the **T**ool **Lib**rary, version 2.00. The reason there are many files is because the archive is very large, and downloading many small pieces is less prone to error than downloading a single large file. Join the pieces together again with '`cat`' (or MS-DOS `copy` on Windows) to re-create the original archive:

```
cat tc_160.000 tc_160.001  … > tc_160.tgz          (Linux)
        or
copy /b tc_160.000 + tc_160.001  …  tc_160.tgz     (MS-DOS)
```

Archives are distributed in "`.tgz`" format (the extension can also be '`.tar.gz`'). This is a **tar** file that has been compressed with **gzip**. (A tar file is a way of combining many files into one big file, without compression. Gzip is a compression utility that works on single files only)

To extract a "`.tgz`" file, use a command like:

```
tar –C /usr/local -xvzf tc_160.tgz
```

This will extract the archive tc_160.tgz into the directory /usr/local. If you are using Windows, Winzip can decompress "`.tgz`" files.

Installing the tools

All development tools and libraries are contained under the directory `/usr/local/sce`. Unpack the latest `tc_xxx.tgz` and `tlib_xxx.tgz` files to `/usr/local/`:

```
tar –C /usr/local –xvzf tc_xxx.tgz
tar –C /usr/local –xvzf tlib_xxx.tgz
```

Setting up the paths

The executable tools lie in three different directories. You must add these to your shell path to be able to execute them from the command line. These directories are:

```
/usr/local/sce/bin              (Sony TOOL connection programs)
/usr/local/sce/ee/gcc/bin       (EE compiler tools)
/usr/local/sce/iop/gcc/bin      (IOP compiler tools)
```

You can add these to your current path using the command:

```
setenv PATH ${PATH}:/usr/local/sce/bin:/usr/local/sce/ee/gcc/bin:/usr/local/sce/iop/gcc/bin
```

(tcsh/csh users)

or

```
PATH=$PATH:/usr/local/sce/bin:/usr/local/sce/ee/gcc/bin:/usr/local/sce/iop/gcc/bin
export PATH
```

(bash users)

Put the appropriate command into your startup script so that this is performed every time you log in.

Read the setup files in `/usr/local/sce/1st_read/` for more information.

There are two important chips inside the PS2 called the **EE** and the **IOP**. Each of these chips has their own separate compilers, libraries and samples, and these are kept separately in the `/usr/local/sce/ee` and `/usr/local/sce/iop` directories respectively.

Layout of the development tools

`/usr/local/sce/` - Everything related to PS2 development is stored under this directory.

- `1st_read` – This directory contains ReadMe files concerning how to set up your development environment, gcc (the compiler), changes since the last release, and other important information.
- `bin` – All the Sony tools used to communicate with the TOOL.
- `ee` – Compilers and samples for the EE processor.
- `iop` - Compilers and samples for the IOP chip.
- `common` – files that both the EE and IOP compilers need. This is mostly library header files.
- `doc` – Some documentation is stored here (not all of it). This includes technical notes, kernel references, library references and information on how to use the Sony tools.
- `rpm` – Stands for Redhat Package Manager, the file format used to install applications on RedHat Linux machines. The PC inside the TOOL runs Redhat Linux, and some of the software occasionally needs to be updated. This is the directory where those packages are stored, and the TOOL can be updated by using the web administration interface to transfer RPM files from this directory to the TOOL.
- `tools` – These tools are artist creation and manipulation tools, like sound/video processors or viewers.

`/usr/local/sce/ee` – Everything under this directory is concerned with the EE chip.

- `gcc` – the compiler tools, manuals and standard include files for the EE.
- `include` – the library header files for the PlayStation 2 specific libraries
- `lib` – the PlayStation 2 development libraries.
- `sample` – many different samples showing how to use the various components of the system
- `src` – source code to a selected subset of the PlayStation 2 libraries

`/usr/local/sce/iop` – Everything under this directory is concerned with the IOP chip.

- `gcc` – the compiler tools, manuals and standard include files for the IOP.
- `install` – extra libraries and include files that must be copied to the correct directory when package is installed for the first time.
- `modules` – IOP modules (the equivalent of IOP programs). Some of these are loaded as standard whenever the TOOL is reset.
- `sample` – many different samples showing how to use the various components of the IOP.
- `src` – source code to a selected subset of IOP modules.
- `util` – Various utilities for IOP development.

Installing the IOP libraries

To install the IOP compiler, unpack the latest tool chain to `/usr/local/`. The IOP compiler tools will be placed in `/usr/local/sce/iop/gcc/bin/`.

However, to avoid infringing certain aspects of the GNU distribution agreement, some components of the IOP libraries must be manually installed into the IOP directories. These components are contained in the tool library distribution, and are contained in `/usr/local/sce/iop/install`.

After unpacking the tool chain **and** the tool libraries, you must copy the contents of `/usr/local/sce/iop/install/include` into `/usr/local/sce/iop/gcc/mipsel-scei-elfl/include`, and copy the contents of `/usr/local/sce/iop/install/lib` into `/usr/local/sce/iop/gcc/mipsel-scei-elfl/lib`.

You can perform this with:

```
cd /usr/local/sce/iop
cp –r install/* gcc/mipsel-scei-elfl
```

Incremental library updates

Sometimes library releases are incremental. Instead of releasing the entire library to download again, only the changes are released. For some releases, this means installing a new flash ROM (see below).

When an incremental release is obtained, ensure that all previous releases have been installed, and simply extract the new release into `/usr/local/`. It will add and replace the appropriate files.

Installing TOOL packages

The TOOL contains a PC that handles the network connections and allows development PCs to communicate with the PlayStation 2. It runs various servers and other support progams, and these sometimes need to be upgraded.

These packages will be contained in `/usr/local/sce/rpm`. The TOOL web administration interface can be used to install the packages. The package manager on the TOOL will use FTP to obtain new RPM files. Therefore it is necessary to have a FTP server set up so that the TOOL can connect to it and obtain the RPM.

The web interface will attempt to log in and obtain the RPM file. If you have trouble upgrading a package, it is probably an FTP problem. Try using ftp from another machine to obtain the RPM file, as this will give you a better indication of where the FTP process is failing.

There are some packages that usually need to be installed.

- `dsnetm` handles communication between the development tools and the PlayStation 2. The RPM is found in `/usr/local/sce/rpm`. You will get connection errors if the version of `dsnetm` currently installed is a lower version than what the tools expect.

- `gstool` displays the initial boot up screen. If you encounter a black screen instead of the IP address display, then this needs to be upgraded. This can be obtained from the developer website under `Tools-Programming/T10000_Administration/gst_14.zip`.

- `pstool` fixes a Y2K bug within the web interface. This can be obtained from the website under `Tools-Programming/T10000_Administration/pst_12_1.zip`.

Communicating with the TOOL

Once you have all the software installed and the execution paths set up, you should try contacting your TOOL over the network using the development tools. After your TOOL is configured with an IP address, try running the following:

```
dsiping -d x.x.x.x
```

(where `x.x.x.x` is the name or IP address of your TOOL). `dsiping` is in `/usr/local/sce/bin`, and is used to 'ping' the IOP to see if it is contactable.

You should see a display like "`32 bytes from IOP: seq=0 time=0.711 ms`". Press Control-C to exit.

Passing the IP address to development tools

All of the Sony development tools must be passed the IP address of the TOOL they will connect to. You can specify the address to any '`ds…`' command (`dsedb`, `dsflash`, etc) using the `-d` option, as used above in the `dsiping` example. For convenience, if you set the environment variable `DSNETM` with the IP address or DNS hostname of the TOOL, then you do not have to specify the address on the command line.

To set the `DSNETM` environment variable, use:

```
setenv DSNETM my_TOOL_IP_address_or_name                (csh/tcsh)
            or
DSNETM=my_TOOL_IP_address_or_name ; export DSNETM      (bash)
```

In the examples below, it is assumed that the `DSNETM` variable is set, and therefore the examples do not explicitly specify the TOOL IP address to connect to.

Changing the flash ROM in the TOOL

The TOOL has a built-in ROM that contains the IOP and EE operating systems. This ROM is "flashable", which means that it's possible to update the ROM over the network. Generally, each new release of the tool libraries comes with a new flash ROM, which fixes bugs and adds functionality.

You need to make sure that your TOOL ROM is 'flashed' to the version most appropriate to the version of the tool libraries that you're using. If you've unpacked the tool library to `/usr/local`, you'll find the latest flash ROM file in the `/usr/local` directory. It will be called something like `t10000-rel16.bin` or `t10000-1020.bin`.

To flash your TOOL, run the '`dsflash`' program, passing the path to the appropriate ROM file as the argument.

It is **VERY** important that the flashing process is not interrupted. It is recommended that you inform colleagues before flashing the TOOL, as it is possible for someone to inadvertently interrupt the process. This may corrupt the ROM, rendering the TOOL unusable.

Compiling and running the samples

Compiling EE samples

All samples for the EE are kept under `/usr/local/sce/ee/sample`. Change to this directory and type '`make`'. This directory contains a Makefile that will recurse through all the sub directories and compile all the samples.

Compiling IOP samples

Some of the samples require IOP modules to work. To make IOP modules, you must change directory to `/usr/local/sce/iop/sample`, and enter every directory manually and type '`make`'. Alternatively, you can type:

```
find /usr/local/sce/iop/sample –name Makefile|sed –e s/Makefile//|xargs –n 1 make -C
```

This will find any directories that have a Makefile, change directory to those directories, and run `make`.

Running the samples

To run EE programs, you use '`dsedb`'. To test whether you can run a program on your TOOL, compile the EE samples, then:

```
        dsedb -r run /usr/local/sce/ee/sample/vu1/hako/sample.elf
```

You should see a rotating cube on the screen. Press Ctrl-C to quit, and some debug information will be displayed, and you'll be returned to the shell command prompt.

About the compilers

Sony uses the free GNU tools to create PlayStation 2 software. (See `www.gnu.org` for more information on GNU tools). The compiler is called GCC (**G**NU **C** **C**ompiler). It can also compile C++ and assembly language programs. We'll discuss the tools in more depth later.

If you need information on the specifics of gcc or make (such as command line options and what they mean), then you can access online manuals at `http://www.cl.cam.ac.uk/texinfodoc/dir.html` . There are also info files in `/usr/local/sce/ee/gcc/info`. (Run `info` in this directory to access them)

Where to go from here

Now that you can load and run EE programs, we can begin learning about the system and how to program it. In the next section, we'll discuss the architecture of the PS2.

If you want to run the other samples, you can load them in the same way that you loaded the `hako` sample (note that the filename of the `.elf` executable might be different for other samples). Some samples may require IOP modules to be compiled before they can be run.

If you have access to the support website, then you'll be able to download many more samples that demonstrate different techniques on the PS2.

Troubleshooting

When I boot the TOOL, the TV only displays a black screen, not the IP address information screen.

If you can still ping the TOOL, then make sure the `gstool` package is installed. See Installing TOOL packages for more information.

The web administration interface cannot seem to obtain a list of available packages.

This is usually a network or FTP problem, rather than a problem with the TOOL. Try using the IP address of the FTP server where the RPMs are located instead of the name. Try FTPing the packages from another computer, as this will give you a better idea of what the problem is.

We have forgotten our Web Administrator's password.

The TOOL will have to sent back to Developer Support for re-initialisation.

The TOOL has an IP address, but we cannot ping it.

There are two general causes. The first is physical – for reasons which are not clear, some TOOLs have problems communicating with PCs over Ethernet switches (not hubs). Try using a hub instead of a switch. The second is configuration, and is probably caused by the PC attempting the ping not being on the same subnet as the TOOL. Temporarily change the PC to be on the same subnet as the TOOL (i.e. set the first 3 numbers of the PC's IP address to that of the TOOL's). This will enable you to contact the TOOL over the web interface, allowing you to change the TOOL's IP address to something more suitable.

The development tools exit with an error about the version of `dsnetm`.

Install the latest version of the `dsnetm` RPM onto the TOOL. See "Installing TOOL packages."

The development tools exit with an error about the incorrect version of the flash ROM.

Flash the TOOL with the version of the flash ROM appropriate to the version of the development TOOLs you are using. See "Flashing the devkit" for more details.

When updating the flash ROM, the process was interrupted, and it is no longer possible to flash the ROM.

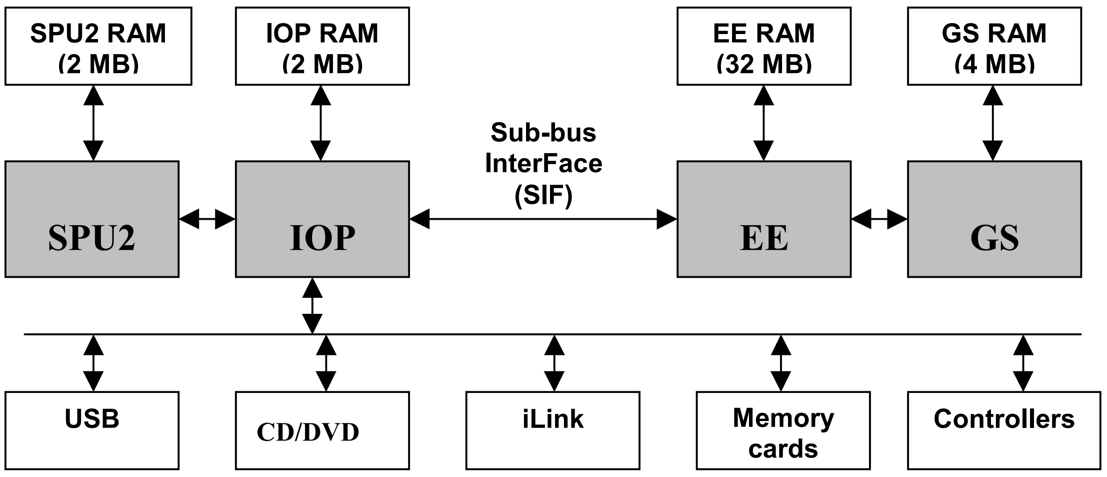
Contact Developer Support who will walk you through re-initialising the ROM.

Many errors are output when we try to compile the samples.

If you have problems compiling the samples, it is probably because the Makefiles expect the development libraries and header files to be in specific places. Make sure the development software is installed under `/usr/local/sce`. Also check your execution paths are set up correctly, so that the Makefiles are able to access the programs they need to compile the samples.

Section B - Inside the black box

This section covers the important aspects of the internal architecture of the PS2. Most of this is covered in more detail by the black EE overview manual shipped with your TOOL.



There are 4 chips that are the most important components of the system, highlighted above in a darker shade. Each chip has its own area of RAM so it can work independently of the other chips.

SPU2

The SPU2 is the sound chip. It has 48 channel ADCPM (a Sony proprietary compression format that reduces sound to a quarter of its original size), 2 MB of RAM, and some PCM (i.e, raw uncompressed WAV) channels. In general, you don't program the SPU2 directly, there are calls made by the IOP that activate its features, and transfer data into the SPU2 RAM.

It is called SPU2 because it is essentially 2 copies of the original PlayStation SPU chip.

IOP

IOP stands for Input/Output Processor. It is the same as the original PlayStation R3000 MIPS chip. It runs at 37Mhz, and has 2MB of RAM. The TOOL version has 8MB.

The IOP handles all the input and output to the peripherals. These include the iLink port, the USB port, the memory cards, controllers, and the CD/DVD unit. For the TOOL version, it also handles all communication with the PC inside the TOOL. Any other chip that wants to communicate with an external peripheral must do so via the IOP. The IOP is the only way data can get in or out of the system. This means you can't load CD/DVD data directly into EE memory – it has to go through the IOP first.

The IOP has a custom pre-emptive multi-tasking operating system that it runs all the time. Unlike the original PlayStation, the developer does not have complete control of the chip. Instead, programs called 'modules' are created that the IOP can run. These have the file extension '`.irx`', and act like separate threads.

A typical module would be a USB device handler, or perhaps something as complex as a TCP/IP stack. You don't have low-level access to all the peripherals that are controlled by the IOP, so modules you write have to make calls to the IOP OS to handle these devices.

The IOP has its own special compiler. It's a version of GCC built to create standard R3000 code (the IOP is a modified MIPS R3000) and create files in the special modular `.IRX` format which the IOP OS expects.

The IOP is connected to the EE by a special separate bus called the SIF (Sub bus InterFace).

GS

The GS (Graphics Synthesiser) is the very very powerful graphics chip that has massive bandwidth to the EE and has a huge fill rate. It has 4MB of RAM that is stored onboard the chip, so it is much faster to access than normal memory. 4MB may not sound like much, but the phenomenal bandwidth of the GS means that you could transfer about 20MB of data to it *per frame* if you so desired.

The GS features Z-buffering, perspective correct textures, mip-mapping, anti-aliasing and bi/tri-linear filtering.

The GS is 'dumb' in that it has no built-in transform and lighting calculation abilities. The EE performs all transform and lighting calculations, then sends the GS the commands to draw the 2D polygons.

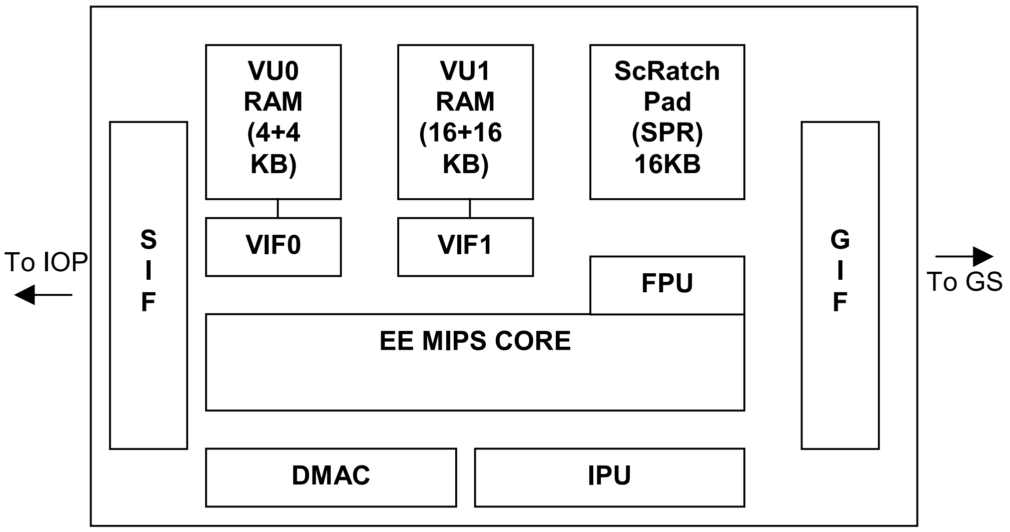
The frame buffers (the buffers that are used to draw/display a screen) and the Z-buffer are all stored in the 4MB of VRAM. There are a few techniques to minimise the amount of space these buffers require in order to leave more room for textures. Although the GS doesn't support compressed textures (apart from textures with palettes), there are multi-pass techniques that can be used to effectively compress textures.

The only chip that can directly access the GS is the EE.

EE

The EE (Emotion Engine) is the very powerful chip designed to do most of the work. The fact that it can only talk to the IOP, its own RAM and the GS, means it doesn't get interrupted much, and it can get on with its main job of running the main game loop and creating the graphics. The EE is the chip you'll probably spend the most time programming.

It has 32 MB of main RAM, as well as an instruction and data caches and 16KB of scratchpad RAM. The TOOL version has 128MB of main RAM. The EE is also directly connected to the GS, and is the only chip that can talk to the GS. This direct connection means that the EE can transfer data and commands to the GS at a massive rate (2GB/s).



This is a simplistic diagram of the EE. It is not complete, and does not show all the interconnections, but it does show the important components.

The EE is complex. It is comprised of a MIPS processor core, an Image Processing Unit (IPU) for helping decode JPEG images and MPEG streams, a Graphics InterFace (GIF) for talking to the GS, and two Vector Units, which are essentially two separate processors with their own RAM and assembly language. The Vector Units are where the power of the system lies, so they're very very important to understand.

There are effectively 3 separate processors on the EE chip. Using these processors in parallel is the key to the power of the system.

EE core

The main processor on the EE the R5900 MIPS core (usually referred to as the EE core), a custom MIPS III architecture processor. This processor has 32 registers that are 128-bits wide each, and there are opcodes to operate on 32, 64 or 128 bits at a time. The instructions designed to operate on 128 bits are called the multimedia instructions, and are a non-standard part of the MIPS architecture.

The EE core is unusual in that it is the first MIPS processor to be superscalar. That is, there is more than one pipeline in the core, so it is possible for two consecutive instructions to travel through the pipeline side by side. This gives greater throughput and hence better performance.

The EE core has its own compiler (called `ee-gcc`), which is simply the standard MIPS gcc compiler modified to handle the extra custom 128 bit instructions.

It also has a standard MIPS FPU (Floating Point Unit) to handle single precision (32-bit) floating point calculations. The EE core's FPU does not handle the bulk of floating point operations however, as these can be performed much faster with the Vector Units.

VUs

The other two processors embedded inside the EE are almost identical in design. These are the Vector Units (or VUs as they're more commonly referred to), called VU0 and VU1. The two Vector Units were each designed for slightly different purposes, so they differ in a few small ways. The VUs each have their own code and data memory, built in to the EE.

The Vector Units are extremely powerful. They designed very similarly to the MIPS philosophy. That is, each instruction is very simple. There are instructions to move data between registers and VU memory, and there are instructions to perform calculations on data held in registers. There are no instructions that perform calculations on data stored in memory. Each VU cannot access the memory outside of its own VU memory.

A VU has thirty-two 128-bit registers, which each hold 4 single precision floating point values [representing X, Y, Z and W]. There are 4 floating point units in a VU that can all work in parallel on a register or registers. For example:

```
MUL.xyzw VF20, VF21, VF07
```

This instruction will multiply the 4 elements (X,Y,Z,W) of register VF07 by the 4 corresponding elements of register VF21, and put the result in register VF20. This is equivalent to performing:

```
VF20.x = VF21.x * VF07.x
VF20.y = VF21.y * VF07.y
VF20.z = VF21.z * VF07.z
VF20.w = VF21.w * VF07.w
```

Each VU also has an integer unit, which has 16 integer registers. These are usually used for holding pointers, indexing data in memory and for loop counters.

Each vector unit can be programmed and run independently. VU0 has 4KB for code and 4KB for data. VU1 has 16KB for code and 16KB for data. Although they are functionally very similar, the connectivity of the units to the EE and the GS means they are suited to different jobs. VU1 is connected directly to the graphics chip, so it's main purpose is creating/transforming 3D polygons and sending draw lists to the GS. The VU0 is only connected to the EE (and the VU0 has less RAM than the VU1), so it's good for performing tasks like physics calculation or preparing matrices to send to the VU1.

The VUs are a little different to processors you may have seen in two important ways:

Firstly, they don't necessarily run all the time. Most processors never stop executing instructions once they start, even if that means they are just looping around waiting for some event. However, once a VU program is started, the VU program can halt itself, or can be halted by the EE core. Halting a VU program is just like putting it into suspended animation – it can be restarted externally and continue as if nothing had happened. This allows you to write VU programs that can be activated, left to process data and then halt, all while the EE core is performing other operations.

As an analogy, you can consider the VUs to be like an automatic washing machine. A load of washing [data] is put in, it's activated, then the user goes and does something more interesting. When the user [EE core] is ready and the washing machine has finished, they can take their newly processed washing [data] out, put a new load in, and set it off again. Just like a washing machine, if the user is still performing tasks when it finishes, it happily waits until the user is ready to empty it.

The second difference that you'll see very quickly if you look at any VU programs (extension '`.dsm`') is that there are **two** instructions on every line. This is because the VUs have a Very Long Instruction Word (VLIW) architecture. The leftmost instruction is for the floating point units, and the rightmost instruction is for the integer unit. As each instruction uses a different unit, both can be executed at the same time. The left instructions do all the vector operations, and the integer units take care of more mundane tasks like branching and data indexing.

A VU program is created using the VU assembler (`ee-dvp-as`). This creates a binary file containing the VU microcode, which is transferred into EE RAM along with the main program. The DMAC can be used to upload programs and data to VU memory, and can also be used to activate VU programs.

DMAC

The DMAC is the DMA Controller. DMA stands for Direct Memory Access. The DMAC is an extremely important part of the EE, and it's very important to understand what it does and how it works.

Essentially, the DMAC copies data between EE main memory/scratchpad and other EE components. The DMAC can be considered to be a 'pigeonhole stuffer'. It reads a specified area or areas of memory, and stuffs them as fast as it can through a pigeonhole (interface) to an EE component. The component receives a continuous stream of data through its particular pigeonhole. (The DMAC can also work in reverse for some components, receiving data through the pigeonhole and placing it in memory)

The EE core sets up a DMA transfer by setting some memory mapped registers, then leaves the DMAC to perform the copying while it continues with other tasks.

There are 10 DMA channels, and each channel has a specific EE component that it talks to. Usually the source is main memory and the destination is an interface to another processor. The DMAC is the way data is transferred between the IOP, VUs, and the GS.

For example, if you wanted to send data directly to the GS, you'd use channel DMA channel 2. This channel is specifically for moving data to and from the GIF, which is the EE component that talks to the GS. You'd tell the DMAC (by setting some registers) where your data was and how large it was, then activate the DMAC and your data would get transferred via the GIF to the GS. Hopefully it contains some useful drawing commands for the GS to use!

The DMAC, at its simplest level, can be told the address/length of a set of data and a destination, and be left to transfer that data to its destination. However, it can also use special codes embedded in the data called "DMA tags" which are ways of automatically telling the DMAC to transfer multiple different areas of memory to a destination, all in one transfer with no EE core intervention.

The DMA can also be used to control a VU. If you embed codes called "VIF codes" in the data that the DMA transfers, it can be used to upload programs to the VUs, start those programs, and the DMA can even stall until the VU program is complete. This allows you to send large streams of programs and data to the VUs - the DMA will 'feed' the VU with data until there's no more data to process, all without EE core intervention.

VIF

Each VU has a **V**ector Unit **I**nter**F**ace (VIF). This is the interface that the DMA sends data to when it wishes to transfer data to VU memory. By embedding special VIF codes in the data being transferred to the VIF, the VIF can perform certain operations including setting the address that the data goes to, decompressing data, stalling the transfer until the VU program is complete, or starting VU programs.

GIF

The GIF is the **G**raphics synthesiser **I**nter**F**ace. This component talks directly to the GS, and arbitrates between other EE components (such as VU1, VIF1 and the DMA) that wish to transfer data to the GS.

SPR

The SPR (Scratch Pad RAM) is 16KB of memory on the EE, that only the EE core and the DMAC have access to. It's useful because it's so fast (single cycle access). It can be better than a cache because access to it is explicit (unlike a cache), and therefore it can be faster to build certain structures on SPR rather than build them in main RAM.

Summary

We've covered most of the important components of the system. To summarise:

- The SPU2 handles sound, and is controlled by the IOP.
- The IOP handles all peripherals. If the EE, GS or SPU2 require data from the CD or some other peripheral, they have to use the IOP to retrieve it for them.
- The GS draws all the polygons.
- The EE is the powerful processor that creates and transforms the polygons, then sends them to the GS. It's also used to run the main game code.

To summarise the EE:

- It consists of many components, all of which can run in parallel.
- The EE core is a MIPS processor that can be considered the 'master' component of the EE (i.e. it controls the other components of the EE).
- The DMAC moves data around the EE and to other peripherals.
- The VUs can be considered separate processors that do the bulk of the floating point work.
- The VIF, SIF and GIF are interfaces to other peripherals. The VIF is the interface to the VUs, the SIF is the interface to the IOP, and the GIF is the interface to the GS.

What to do now

You will have received a set of large black manuals with your development kit. The latest versions of these are also available for download at the SCEE website.

It is advised that you read the EE Overview manual, and glance over the PS2 library overviews in `/usr/local/sce/doc/`.

Section C - The tool chain

The tool chain is the set of applications used to create and run PlayStation 2 programs on the TOOL.

There are 5 important applications in the tool chain:

- **ee-gcc** – This can compile and link C, C++ and assembly language programs for the EE MIPS core (which is why it is prefixed with 'ee').
- **iop-gcc** – Similar to ee-gcc, iop-gcc can compile and link C, C++ and assembly language modules for the IOP.
- **dsedb** – Used to upload, run and debug programs on the EE core.
- **dsidb** – Used to upload, run and debug modules running on the IOP.
- **ee-dvp-as** – DMA, VIF and VU programs are assembled with this tool.

There are also other tools for managing object files.

Compiling and running an EE program

Most programmers will be starting to work on the EE. Here is the traditional example of creating and running a Hello World program on the EE.

```
#include <stdio.h>
int main(void) {
  printf("Hello World\n");
  return 0;
}
```

Type the above program into a file called `helloworld.c`. Compile it into a `.ELF` file with the following 3 commands (the arguments should all be on one line):

```
ee-gcc -c -xassembler-with-cpp -o crt0.o /usr/local/sce/ee/lib/crt0.s

ee-gcc -fno-common -I/usr/local/sce/ee/include -c helloworld.c -o helloworld.o

ee-gcc -o helloworld.elf -T /usr/local/sce/ee/lib/app.cmd crt0.o helloworld.o
-nostartfiles -L/usr/local/sce/ee/lib
```

The first two commands create the necessary objects. The last command links them together to make the `helloworld.elf` executable.

There is a reason why we broke the compilation into 3 stages. Unlike other programs that run on conventional operating systems, your PS2 program has complete control of the EE. That level of control even boils down to how your program starts up, and where your program is loaded into memory. So we have to specify these parameters explicitly.

- The `crt0.s` file is a startup file that initialises the stack, the heap and the BSS section. It replaces the standard startup code that gcc would normally link in. `crt0.s` is a standard file and wouldn't be changed under normal circumstances.

- The `app.cmd` file is a linker script that tells the linker some information about section alignment, the stack size, and where your program should be loaded in memory.

You may also have noticed that although we were performing 3 distinct operations (assembling, C compiling and linking), we used `ee-gcc` each time. That's because `ee-gcc` is intelligent enough to figure out which one of its partner applications is required and calls the appropriate application.

Running the program

To run this program on the EE, we use the program `dsedb` to upload and run it. Type:

```
dsedb -d x.x.x.x -r run helloworld.elf
```

(where `x.x.x.x` is the IP address or name of your development kit. You can omit this if you've set the IP address up in the `DSNETM` variable. In the examples that follow, it is assumed that you've done this.)

You should see something like:

```
***Resetting...

EE DECI2 Manager version 0.06 May 11 2000 18:08:48
  CPUID=2e14, BoardID=4126, ROMGEN=2000-1019, 128M

Loading program (address=0x00100000 size=0x0000c738) ...
Loading 491 symbols...
Entry address = 0x00100008
GP value      = 0x00114770
Hello World
*** End of Program
*** retval=0x00000000
```

The 3rd line from the bottom contains the output of the program.

The output in detail

Here is the same output, covered step by step.

```
***Resetting...
```

This line shows `dsedb` is trying to gain control of the TOOL. Since the TOOL is a network device, other people may be using it. The TOOL only allows one `dsedb` session to connect to it at a time. If you get the message "`cannot connect`", then someone else is currently running another program on the TOOL, and you will have to wait for them to quit their program before you can run yours.

When you successfully connect, the IOP and EE are reset to a pre-defined state.

```
EE DECI2 Manager version 0.06 May 11 2000 18:08:48
  CPUID=2e14, BoardID=4126, ROMGEN=2000-1019, 128M
```

DECI2 is the protocol used to exchange data and programs with the TOOL, and can also be used to start and debug programs. The IOP and EE run a special DECI2 manager to handle all this, and these lines let you know the version that's currently running. The second line also tells you the version of the CPU and the board, and the version of the flash ROM (the ROMGEN number is the date of the build in YYYY-MMDD format). Finally, the 128M tells you how much memory the EE has, which will always be 128 megabytes.

```
Loading program (address=0x00100000 size=0x0000c738) ...
Loading 491 symbols...
Entry address = 0x00100008
```

These lines say that the program is being loaded to address 0x0010000. This is the standard address to put programs, and it is specified in the `app.cmd` file. ROM kernel data is contained below this address, so it is not possible to place any of your program or data below address 0x00100000.

The output states that the Entry Address is 0x00100008, which is the address where the program will start to run. This is where the code from `crt0.s` is placed. After the main setup has been performed by `crt0.s`, the `main()` function is called.

```
GP value      = 0x00114770
```

GP is the Global Pointer, which is part of the conventional MIPS register usage. Generally, you don't have to worry about the Global Pointer.

```
Hello World
```

The output of the program. Not much to say about this, but there are a couple of tips about printing.

- If you print a string that does not have a newline ('`\n`')character at the end, then it will not be flushed until a newline is received. If the program ends with characters still in the output buffer, those characters will be lost.

- If you create a program that generates a large amount of `printfs`, it is possible that internal buffers can overflow, which does not crash the program, but does mean that sometimes output is lost. For example, run a program which prints out the values from 1 to 20000 on separate lines. Examine the output and you'll probably find that some lines are missing. To get around this, you either have to reduce the amount of output you're sending, reduce the speed you're sending it, or use other means to get the data out (Saving to disk is always reliable).

```
*** End of Program
*** retval=0x00000000
```

This shows that the program has finished, and `dsedb` displays the return value of '`main()`', which was zero. If you call the function `exit()` then the program will immediately end, and `dsedb` will print the value of `exit()`'s parameter.

More about `dsedb`

The tool `dsedb` loads EE programs into EE memory and runs them. It can also be used to debug programs. For example, at the command line type:

```
dsedb
```

This drops you into `dsedb` command mode. Type '`help`' to get a full list of commands. Here are some examples of command mode operations.

- Load your program with '`pload helloworld.elf`'.
- Disassemble the main function with '`di main`'.
- Set breakpoints with '`bp <address or label>`'.
- Run the program with '`run`'.

You can also examine and change memory (including VU memory).

Using `dsidb`

`dsidb` is very similar to `dsedb`. For example, change directory to `/usr/local/sce/iop/sample/hello` and run `make`. This will use `iop-gcc` to compile and link `hello.irx`, which is an IOP module that prints "`Hello!`" and all of its arguments. To run it, type:

```
dsidb
```

then, at the prompt:

```
reset 0 2
mstart hello.irx abcd efgh
```

You will see:

```
Loading 24 symbols...
Hello !
  argv[0] = host1:hello.irx
  argv[1] = abcd
  argv[2] = efgh
```

You may ask why we had to reset with specific arguments before running the program. This is because `dsidb` will automatically load in some support modules, which we don't want for this example. The `reset 0 2` command will not load in any extra modules.

Using `ee-dvp-as`

We'll cover this in a later part, as you need a bit more background about the VIFs before we go into this tool.

Do I have to know assembly language to program the PS2?

Yes, you do, if you want to program the VUs, which is necessary if you are working on any graphics or physics code. There is no C compiler for the VUs, so you have to work in assembly language.

However the IOP and EE core have C compilers, so programming in assembler is not essential. But it is very useful to have background knowledge of the MIPS processor architecture, and the register usage conventions that C uses on a MIPS processor. A background knowledge of what the compiler is doing can help you drastically improve your code's efficiency.

It will also help you understand `dsedb's` output if your program crashes.

Books about MIPS programming

As the Toshiba R5900 is a custom chip, there are no third party books specifically about it. However, it is very similar to traditional MIPS architecture. We recommend two books for those wishing to learn more about MIPS.

1. "MIPS Programming" (also called "See MIPS Run") by Dominic Sweetman. Morgan Kaufmann; ISBN: 1558604103. This is an excellent guide and reference to the MIPS architecture.
2. "The MIPS Programmer's Handbook" by Erin Farquhar, Philip Bunce. Morgan Kaufmann; ISBN: 1558602976. This contains a good reference to the instruction set, including disassembly of the synthetic instructions.

Section D - Example 1 – First steps

There are many aspects to the PlayStation 2 architecture, and it would be very difficult to be an expert in them all. However, new programmers tend to want to experiment with graphical programs before moving on to other parts of the system. For this reason, the next few examples will cover getting polygons on the screen.

An important note!

The PlayStation 2 is complex. But it is also flexible. Many operations can be performed a number of different ways.

E.g. there are at least 3 ways of determining if a VU0 microprogram has finished. You could stall the EE core with a macromode instruction. Or you could poll the VU0 status register. Or you could set the last instruction of the microprogram to cause an interrupt that signalled the EE core program.

E.g. there are 3 ways of sending data to the GS. These are via the VU1, the VIF1 or directly from EE memory.

E.g. there are multiple ways of moving data between the EE core and VU0/VU1. You could use the DMA and VIFs, write the data into VU memory directly, or use the macromode instructions to set VU0 registers.

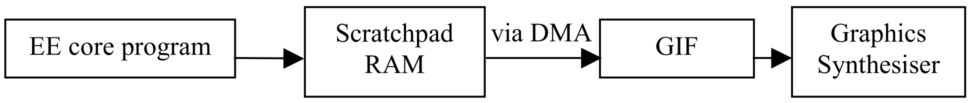
As you can see, there are often many different ways of performing the same task. Each method has advantages and disadvantages. In the example code used here, it is often simplest to use a certain method, even though it is not the 'best' or fastest method. In fact, some of the examples use techniques that you should definitely **not** use, purely for performance reasons! As you become more confident with the system, you can progress to the more advanced methods, and gain better performance by doing so.

Getting a polygon onto the screen

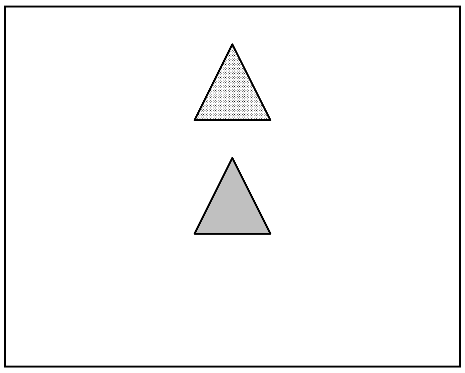
There are multiple ways to draw a polygon on the screen. We will build on the concepts presented in this example.

The easiest and slowest way is to do all the calculations using the EE core.

The main loop of the program builds a list of appropriate Graphics Synthesiser (GS) polygon data on scratchpad RAM. The DMAC transfers this list to the Graphics InterFace (GIF). The GIF transfers the data to the GS where the polygons are drawn.



The result is two triangles on the screen.



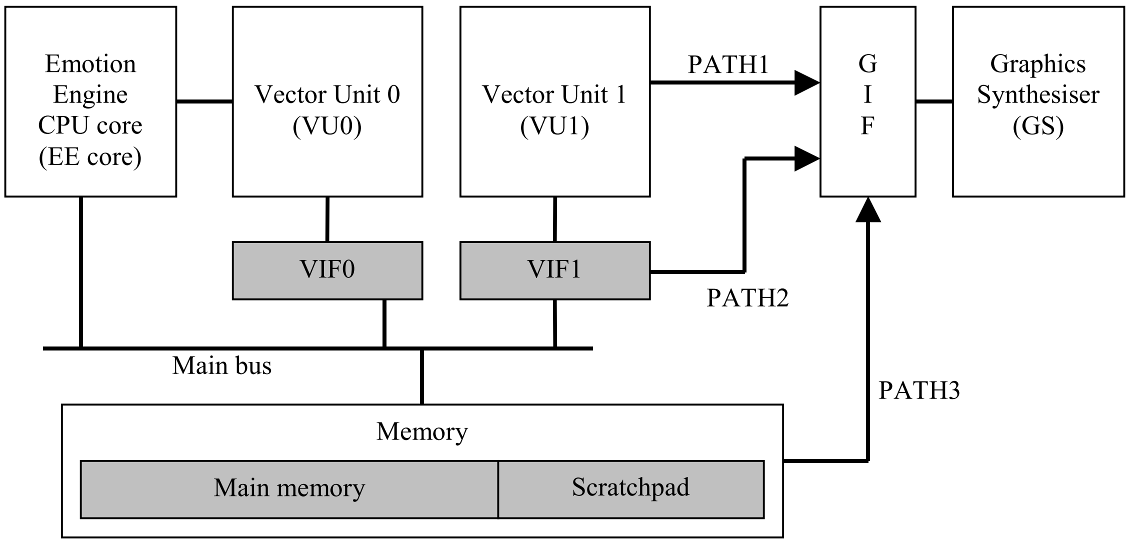
Program output

Fundamentals

Before we begin the code walkthrough, we cover some fundamental concepts important to the way this program works.

General overview

This diagram is an conceptual overview of the architecture components inside the EE. Some components and connections have been removed from this diagram for simplicity.



The GIF is the interface to the GS. It arbitrates between three input sources, PATH1, PATH2 and PATH3. This program creates a GS drawing primitive on scratchpad RAM (SPR) and this is sent to the GIF over PATH3.

DMA

The Direct Memory Access Controller (DMAC) can be considered a co-processor dedicated to moving data between memory and system components.

The EE has 10 DMA channels for moving data around the various parts of the system. Each DMA channel is dedicated to a particular source/destination. Some channels can only transfer one way, others can transfer both ways.

The application primarily uses the DMA channel that moves data from main/scratchpad RAM to the GIF. The GIF will then transfer this data to the GS.

The DMA is controlled by writing values to specific memory locations. The PS2 library '`sceDmaChan'` structure can be used to point to the internal registers used for a particular DMA channel. This structure is initialised the structure with the function `sceDmaGetChan(<channel number>).`

GS and GIF

The Graphics InterFace (GIF) on the EE is the sole interface to the Graphics Synthesiser (GS). All data destined for the GS must pass through the GIF. We transfer data to the GIF using DMA. There are three 'paths' that information can take to flow into the GIF. PATH1 is a direct connection that Vector Unit 1 has with the GIF, PATH2 is a connection from the VIF1 to the GIF, and PATH3 is a DMA channel that transfers data directly from main memory. This application uses PATH3.

The GS has over 70 internal registers, all of which are 64 bits wide and write-only. Writing to these registers controls the behaviour of the GS. For a list of all the GS registers, consult Chapter 7 in the GS User's Manual.

Polygons are drawn in the GS by setting GS registers. For example, a flat triangle is drawn by setting the `PRIM` register (to specify the type of polygon to draw), the `RGBAQ` register (to set the colour), then finally setting the `XYZ2` register three times (to specify the three screen coordinates of the triangle). Once this is transmitted, the GS will have all the information it needs to begin drawing a polygon.

The GS has an internal 7 bit address space for all the registers. Writing to a register updates internal parameters. Additionally, writing to certain registers initiates a drawing operation. These registers are known as *kick* registers, because they 'kick off' an operation.

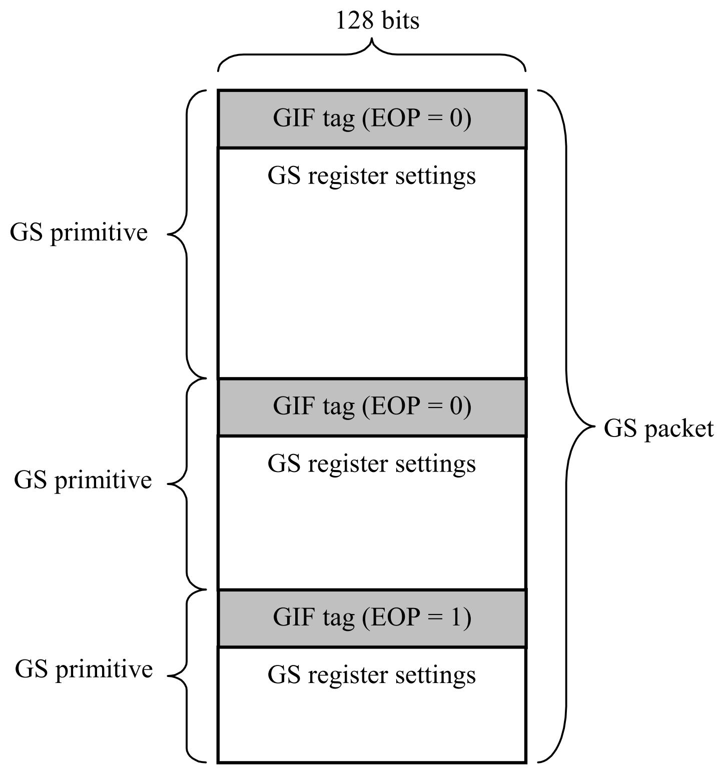
To draw polygons, the application must create a list of registers to set and the data they should contain, and this information is then sent them to the GS.

GIF tags

The information we have to transfer to the GS is a list of 7-bit GS addresses and 64-bit data words. The GIF on the EE is connected to the GS by a 7-bit address bus and 64-bit data bus. This is 71 bits of information per register setting, which unfortunately is not a convenient number. For this reason, the GS addresses are listed using a structure called a *GIF tag*. The GIF tag is sent to the GIF before sending the register data. When the GIF receives the data, it will know which registers to set because they were specified in the GIF tag.

To set GS registers, the program first sends a GIF tag, which lists the registers to be set. Then the program sends the data for each register it wants to set. This combination of GIF tag + data is called a *GS primitive*. This example program will send a single GS primitive to the GIF. This GS primitive will draw two triangles on the screen.

A set of GS primitives is called a *GS packet*. We'll refer to the GS register data as a *GS register setting*.



In these examples, the programs only send a single GS primitive every frame.

A GIF tag is 128 bits in size. 128 bits is also the unit of DMA transfer. 128 bits is equivalent to 16 bytes, and is called a *quadword*. Quadwords and quadword alignment is very common in EE programs.

The GIF tag is embedded in the data that the DMA transfers. Directly following the GIF tag will be the data that fills the GS addresses specified by the GIF tag.

EOP bit

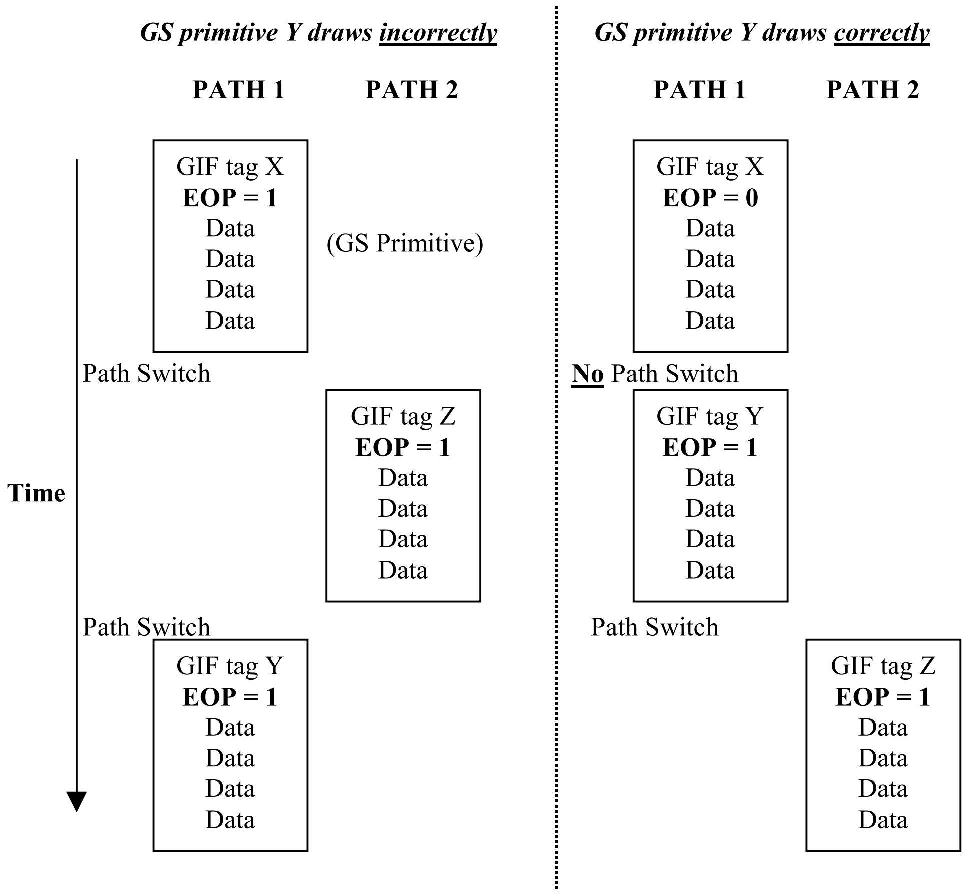
The EOP bit is a field in the GIF tag. It is always set in the last GIF tag in a GS packet.

It is used to allow path switches by the GIF. The GIF has three inputs (PATH1, PATH2 and PATH3) and arbitrates between them to allow each path fair access to the GS. When the GIF changes its source of input from one path to another, this action is called a *path switch*. This is automatic and transparent to the processes generating the three PATH inputs. A path switch may only occur at the end of transferring a GS primitive. However, it sometimes is desirable to suppress a path switch, and the EOP bit performs this function.

When EOP is 1, path switches are allowed. When EOP is 0, path switches are suppressed.

For example, the application may want to use one GS primitive (GIF tag + data) that sets the texture registers, followed by another GS primitive that uses those texture registers. Because the GIF can execute a path switch after processing a GS primitive, the GIF may switch to another path that sends a GS primitive that changes the texture registers again. When the GIF switches back to the original path, the registers will be different to what the new GS primitive was expecting. To explicitly prevent this, the EOP bit is used to indicate when a path switch is acceptable.

This diagram illustrates the problem. GS primitive X is followed by GS primitive Y (both in PATH1), and GS primitive Y relies on register settings performed by GS primitive X. GS primitive Z (in PATH2) writes to the same registers as GS primitive X. In the first situation, the EOP is set to 1 and there is a switch, so GS primitive Y uses incorrect register settings and therefore draws incorrectly. In the second situation, EOP is set to 0 in GS primitive X, so a path switch is suppressed and GS primitive Y is drawn using the correct register settings.



In this example, the program only uses one PATH, so it is not concerned with the setting of the EOP bit. However, it is good practice to set EOP to 1 unless you know you need it set to 0.

Example 1 - Walkthrough

```
#include <eekernel.h>
#include <stdlib.h>
#include <stdio.h>
#include <eeregs.h>
#include <libgraph.h>
#include <libdma.h>
```

The program uses some of the Sony Computer Entertainment (SCE) libraries. These structures and functions are all prefixed with '`sce`'. Although SCE provides many library functions, you are free to replace them with your own (with a few minor exceptions where hardware details are restricted).

Many interesting PlayStation 2 addresses and structures are contained within `eestruct.h`. It is worth examining this file, in `/usr/local/sce/ee/include`.

```
#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 448
#define SPR 0x70000000
```

These are various parameters used to increase the readability of the program. SPR is set to the start address of Scratch Pad RAM. This is 16 KB in size, and is where we will be building our GS packet before it is sent to the GIF.

```
    int buffer;                      // current display buffer
    sceGsDBuff db;                   // Double buffer handler
```

The program uses double buffering, where one buffer is used to display the last complete graphic while the new graphic is being drawn offscreen.

The `buffer` variable stores which buffer (0 or 1) is currently displayed. The `sceGsDBuff` structure holds information about the two double buffers in GS memory used for display and drawing.

GIF tag structure

```
    const u_long myGIFTag[2] = {
        SCE_GIF_SET_TAG(
            2,      // NLOOP
            1,      // EOP
            0,      // PRE
            0,      // PRIM
            1,      // FLG
            5       // NREG
        ),
        0x0000000000055510L
    };
```

Recall that a GIF tag is a quadword that lists the GS registers to set, and the amount of data to follow. This section discusses the structure of the GIF tag.

A GIF tag is 128 bits in size. The code above defines 2 `u_long` integers, which are 64 bits wide each. The first `u_long` is the first half of the GIF tag, and uses a macro to set the appropriate bit fields. The second `u_long` is the second half of the GIF tag, and contains the list of GS registers to set. These two `u_longs` are stored consecutively in memory, and together they make up a complete GIF tag. This will be the first quadword sent to the GIF, which will tell it what type and length of data to follow.

The first 64 bits are created using the macro `SCE_GIF_SET_TAG()` which has the parameters (respectively) `nloop`, `eop`, `pre`, `prim`, `flg` and `nreg`. These are all bit fields within the first half of the GIF tag (see below).

The second 64 bits contain the `REGISTER LIST` field, which is internally split into sixteen fields of 4 bits each. Each field specifies a single GS register to write to.

[4 bits per register address is not enough to specify the entire 7-bit GS address range, but some special conventions are used to address the entire range. These will be explained in the next example.]

GIF tag bit fields

127 64	63 60	59 58	57 47	46		15	14 0
REGISTER LIST	NREG	FLG	PRIM	PRE		EOP	NLOOP

FLG field

The `FLG` field tells the GIF how to interpret the data that is to follow. The DMA sends data to the GIF in units of quadwords (128-bits), and there are 3 different ways the GIF can interpret the quadwords that follow a GIF Tag.

1. PACKED – In this mode, the following quadword is interpreted as one GS register setting. However, 128 bits is too large for a GS register, so the GIF 'packs' the quadword into 64 bits (by discarding parts of the input quadword) and sends it to the GS. This mode is introduced in the next example.
2. REGLIST – This is the simplest mode. The GIF interprets the input quadword as two 64 bit fields. The first 64-bit field is sent to the GS first, and is followed by the second 64-bit field.
3. IMAGE – This is used to transfer texture data. This is not used in this example.

In this example, we are using REGLIST mode (`FLG` = 1).

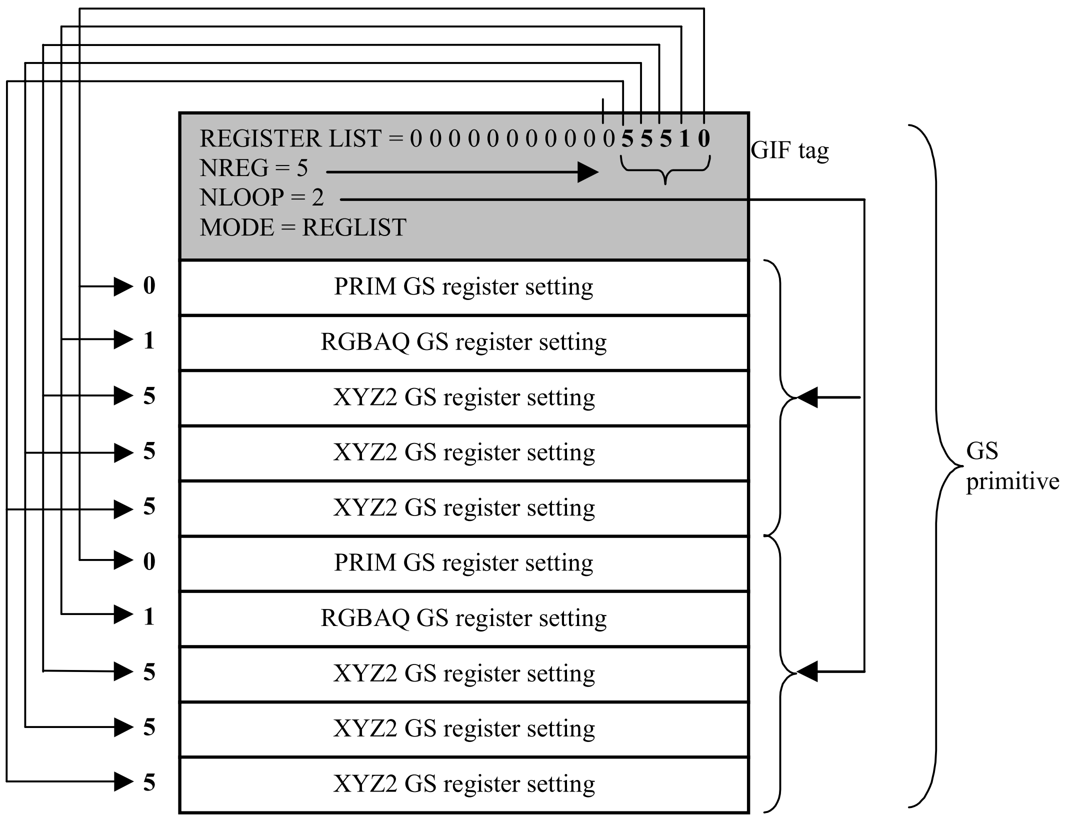
NLOOP, NREG and REGISTER LIST fields

The `REGISTER LIST` specifies which GS registers to write the data into. In this example, we wish to write to GS addresses 0 (`PRIM`), 1 (`RGBAQ`), and 5 (`XYZ2`). `PRIM` sets the type of polygon we want to draw. `RGBAQ` sets the colour. `XYZ2` sets the screen coordinates for the polygon. `XYZ2` must be set 3 times, once for each vertex of the triangle.

This is a total of 5 register settings for one triangle. Therefore, we set `NREG` (Number of REGisters) to 5. The program draws two triangles, so this list of GS registers will be used twice. Therefore we set `NLOOP` to 2.

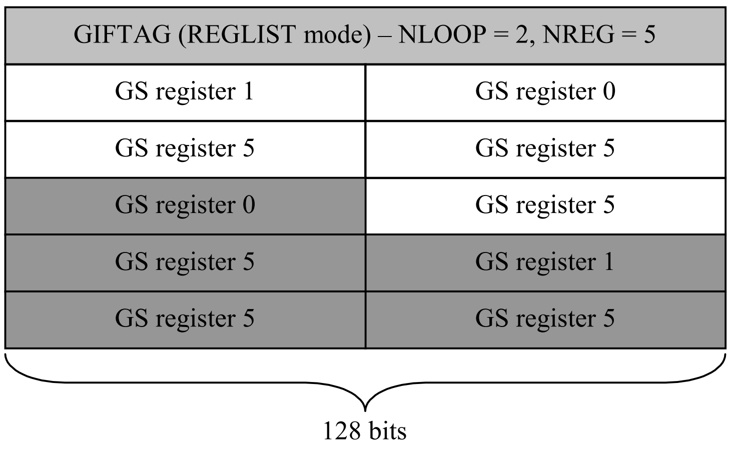
The `REGISTER LIST` is set to `0x55510L`. From right to left, this is the order of the 5 GS addresses we want to set (registers 0, 1, 5, 5, and 5). The 'L' at the end of the number tells the compiler that this should be considered a 64-bit value.

The `NREG` field tells the GIF the number of valid registers in the `REGISTER LIST`. The GIF will expect the data to follow in that order.



In the above diagram, the GIF tag is 128 bits wide, and each GS register setting is 64 bits wide (because we are using REGLIST mode). The number of GS register settings to follow the GIF Tag is calculated by `NREG * NLOOP`.

Below is a diagram of the data structure that the program sends to the GIF. It is arranged into quadwords (128-bits) because that is the DMA unit of transfer, so that's what the GIF will receive. The first quadword is the GIF Tag, describing the data to follow. Then comes the 5 GS register settings for the first triangle, followed by the 5 settings for the second triangle. Each setting is 64 bits in REGLIST mode.



GIFTAG (REGLIST mode) – NLOOP = 2, NREG = 5	
GS register 1	GS register 0
GS register 5	GS register 5
GS register 0	GS register 5
GS register 5	GS register 1
GS register 5	GS register 5

The `PRIM` and `PRE` bits of the GIF tag are not used by this application. They will be covered in the next example.

```
    sceGsResetPath();
    sceDmaReset(1);
```

These calls reset the PATHs that input data into the GIF, and reset the DMA subsystem. These only need to be called once at the start of a program.

```
sceGsResetGraph(
    0,
    SCE_GS_INTERLACE,
    SCE_GS_NTSC,
    SCE_GS_FIELD);
```

This call initialises the graphics subsystem. The parameters for this function are:

`mode`:

0 – performs a complete reset of the GS subsystem
1 – halts all drawing operations and only flushes all existing queued GS instructions.

`inter`:

`SCE_GS_NOINTERLACE` – low vertical resolution.
`SCE_GS_INTERLACE` – high vertical resolution.

`omode` (Output Mode):

`SCE_GS_NTSC` – NTSC television output (60 Hz).
`SCE_GS_PAL` – PAL television output (50 Hz).

`ffmode`:

`SCE_GS_FIELD` – Field based drawing
`SCE_GS_FRAME` – Frame based drawing

These examples use NTSC mode, because it is likely that you will be using a TV that can handle NTSC. If the picture is displayed in black and white or the picture rolls, then use PAL mode instead.

We will cover field and frame based rendering in a later example.

```
sceGsSetDefDBuff(
    &db,
    SCE_GS_PSMCT32,
    SCREEN_WIDTH,
    SCREEN_HEIGHT,
    SCE_GS_ZALWAYS,
    SCE_GS_PSMZ24,
    SCE_GS_CLEAR);
```

This function initialises the screen double buffer structure. The parameters are as follows:

`db`:

This is the address of the `sceGsDBuff` structure that holds double buffer information. It is the structure we are initialising.

`psm`:

This is the bit depth we are using for the display buffer. Valid parameters are:

`SCE_GS_PSMCT32`	32 bit (8 bit RGB, 8 bit alpha)
`SCE_GS_PSMCT24`	24 bit (8 bit RGB)
`SCE_GS_PSMCT16`	16 bit (1 bit alpha, 5 bit RGB)
`SCE_GS_PSMCT16S`	16 bit (1 bit alpha, 5 bit RGB)

`w`:

Width of the screen. The width must be a multiple of 64.

`h`:

Height of the screen buffer. This height depends on whether you are using field based drawing or frame based drawing, which will be covered in a later example. If you are using field based drawing, specify the total height of the screen for both frames. Commonly this is a figure between 448 and 512. When using frame based drawing, this figure is the height of a single buffer, which is commonly 224 or 256. In this code walkthrough we are using field based drawing and use a height value of 448.

`ztest`:

The test mode for the Z-Buffer. When a new pixel is to be drawn, the GS first tests the Z-Buffer value that corresponds to that pixel position to see whether the new colour should replace the existing colour. The following options are available:

SCE_GS_ZNOUSE – never draw pixel, regardless of current Z value
SCE_GS_ZALWAYS – always draw pixel regardless of current Z value
SCE_GS_ZGEQUAL – replaces pixel if new Z value is greater than or equal to the current Z value.
SCE_GS_ZGREATER – replaces pixel if new Z value is greater than the current Z value.

zpsm:

The storage mode for the Z-Buffer. The Z buffer stores the current Z value in 4 different ways:
SCE_GS_PSMZ32 – 32 bits of resolution for the Z buffer.
SCE_GS_PSMZ24 – 24 bits of resolution for the Z buffer.
SCE_GS_PSMZ16 – 16 bits of resolution for the Z buffer. This mode is incompatible with screen storage mode SCE_GS_PSMCT16S.
SCE_GS_PSMZ16S – 16 bits of resolution for the Z buffer. This mode is incompatible with screen storage mode SCE_GS_PSMCT16.

clear:

A flag that specifies whether to clear the screen and Z-Buffer when the double buffer is swapped. A value of 0 means do *not* clear, any other value means clear the screen and Z-Buffer during the sceGsSwapDBuff() call.

For this program, the structure is initialised so that two buffers are used in GS RAM. Each buffer is 640x448 pixels in dimension. A Z-Buffer is also created, but the program does not use a Z-Buffer. This sceGsDBuff structure is used by the sceGsSwapDBuff() function later on in the program.

This structure is part of the standard libraries, however you are permitted to create your own or customise the existing structure. For convenience, we will use the standard libraries to manage our double buffers.

The GS has 4MB of RAM. It holds the frame buffers, Z buffers, and textures. The way memory is allocated to these types of buffers is completely up to the programmer. For simplicities sake, the standard libraries set up the double buffers at the beginning of GS memory (one after the other), and the Z Buffer follows these. The rest of the GS memory is free for use by the programmer.

```
    FlushCache(0);
```

This kernel function flushes the data cache. This is necessary because the sceGsSetDefDBuff() call initialises data that will be transferred by the DMA during the double buffer change. This data is contained in the sceGsDBuff structure, which lies in main memory. Since the DMA does not know about the cache, it is necessary for the main memory to be consistent with the lines in the data cache. If the program creates a structure to be transferred by DMA and writes this data to cached memory, then the main memory may not contain the stored contents until the cache has been flushed. To ensure this, the program flushes the entire data cache.

The parameter specifies the different types of flushes that the function can handle:

0 (WRITEBACK_DCACHE) - Writes back contents of data cache.
1 (INVALIDATE_DCACHE) - Invalidates contents of data cache
2 (INVALIDATE_ICACHE) - Invalidate contents of instruction cache
3 (INVALIDATE_CACHE) - Invalidate contents of both instruction & data cache

Flushing the cache is a **very slow** operation and should be avoided where possible. There are techniques used in later examples that mean the cache does not have to be flushed.

```
    dmaGif = sceDmaGetChan(SCE_DMA_GIF);
```

The DMA channels are controlled by memory mapped registers. Each channel has its own set of registers, and these are arranged in consecutive memory addresses. The sceDmaChan is a structure that is designed to assist access of these registers. When sceDmaGetChan is called, it returns a pointer to the start of the register set for the channel specified. You can then use an indirect reference through structure to access the DMA registers. See eestruct.h for more information on the sceDmaChan structure.

In these examples, we will use library functions to manage our DMA.

```
    spr = (u_long *)SPR;
```

The program starts to build the DMA list to send to the GIF. The `spr` variable is a pointer to an `u_long`, which is 64 bits wide. This pointer is set to address 0x70000000, the beginning of Scratchpad RAM.

We will build the DMA list on scratchpad RAM for three reasons.

1. It saves us having to allocate an area of memory using the `malloc` or `memalign` functions.
2. It is fast to write to (single cycle access).
3. Because it is not cached, the program does not have to flush the data cache before beginning the DMA transfer. As mentioned before, if you are building an area of memory to be transferred by DMA, then you must be sure that all the data has indeed been written back to main memory. This can always be achieved by flushing the cache with FlushCache(0), but this is slow. There is a better way involving using uncached memory, and this is be covered in a later example.

```
    *(spr++) = myGIFTag[0];
    *(spr++) = myGIFTag[1];
```

The first element copied is the GIF tag quadword. This will be followed by the GS register settings, which are each 64 bits wide.

```
    *(spr++) = SCE_GS_SET_PRIM(
                SCE_GS_PRIM_TRI,    // PRIM (Primitive type)
                0,                  // IIP  (Gouraud)
                0,                  // TME  (Textured)
                0,                  // FGE  (Fogging)
                0,                  // ABE  (Alpha Blending)
                0,                  // AA1  (Anti-Aliasing)
                0,                  // FST  (Use ST for texture coords)
                0,                  // CTXT (Context)
                0);                 // FIX  (Fragment control)
```

The first 64 bits of GS register data is destined for the GS `PRIM` register. Writing data to this register will clear the vertex queues (see below) and specifies the type of polygon the GS will draw.

We use macros to create the GS register settings. This lets us avoid cluttering up the code with many bit shifts and brackets. You can examine the macro definitions in `/usr/local/sce/ee/include/eestruct.h`.

The GS can draw points, lines, line strips, triangles, triangle strips, triangle fans and rectangular sprites. We have set it to be `SCE_GS_PRIM_TRI`, which is a plain triangle. For more information on the other fields, consult Section 3.2.2 of the GS users manual.

```
    *(spr++) = SCE_GS_SET_RGBAQ(0x80, 0x00, 0x00, 0x00, 0);
```

The next GS register setting is the RGBAQ register, which sets the colour of the polygon. As usual, RGB stands for Red, Green and Blue, the three components of the polygon colour. 'A' stands for Alpha. Q is part of the STQ triple, used for perspective correct texture mapping. The program does not use any alpha blending or texture mapping, so the settings of the A and Q fields are arbitrary. For this example, the only important parts are the RGB triple.

The RGB values can range from 0 (black) to 255 (full intensity). The above macro parameters will set the red colour to be half intensity, and the other colours to black.

```
    *(spr++) = SCE_GS_SET_XYZ((2048 - 40) << 4, ((2048 + 40) << 4), 0);
    *(spr++) = SCE_GS_SET_XYZ((2048 + 40) << 4, ((2048 + 40) << 4), 0);
    *(spr++) = SCE_GS_SET_XYZ((2048     ) << 4, ((2048 - 40) << 4), 0);
```

These macros set the three XYZ positions of each of the triangle vertices. To explain why the origin appears to be at 2048,2048, and why all the numbers are shifted left by 4, we need to explain the GS co-ordinate system.

GS co-ordinate system

The GS requires an X, Y and Z value for the endpoints of a line or polygon. However, Z only affects the Z buffer, not the co-ordinate system, so we can disregard it for now.

The GS needs X and Y to each be 16 bits wide, in unsigned fixed point 12:4 integer format. This means that the lower 4 bits are the fractional portion, and the upper 12 bits are the integer portion (from 0 to 4095). The GS needs fixed point numbers for pixel positions because the fractional portion is used to calculate pixel coverage for anti-aliasing.

The values used for X and Y are always in units of pixels. This means that the range of both X and Y is 0 to 4095.9375 (which is 4095 + 15/16), creating an area 4096x4096 pixels in dimension. This is called *primitive space*. Primitive space does not physically exist, and does not correspond to an area of memory – it is just a way of thinking about a virtual (X,Y) area where polygons get drawn. **All vertex coordinates are supplied to the GS in terms of primitive space.**

Obviously, it is not possible to directly map all of primitive space onto a drawing buffer, or even to all of GS memory. Primitive space is just too big. It is only realistic to map a *subset* of primitive space to GS memory.

Once the screen dimensions have been decided (in this case, 640x448 pixels), the program needs to map the drawing buffer in GS memory (a 640x448 pixel rectangle) onto some subset of the 4096x4096 pixel primitive space.

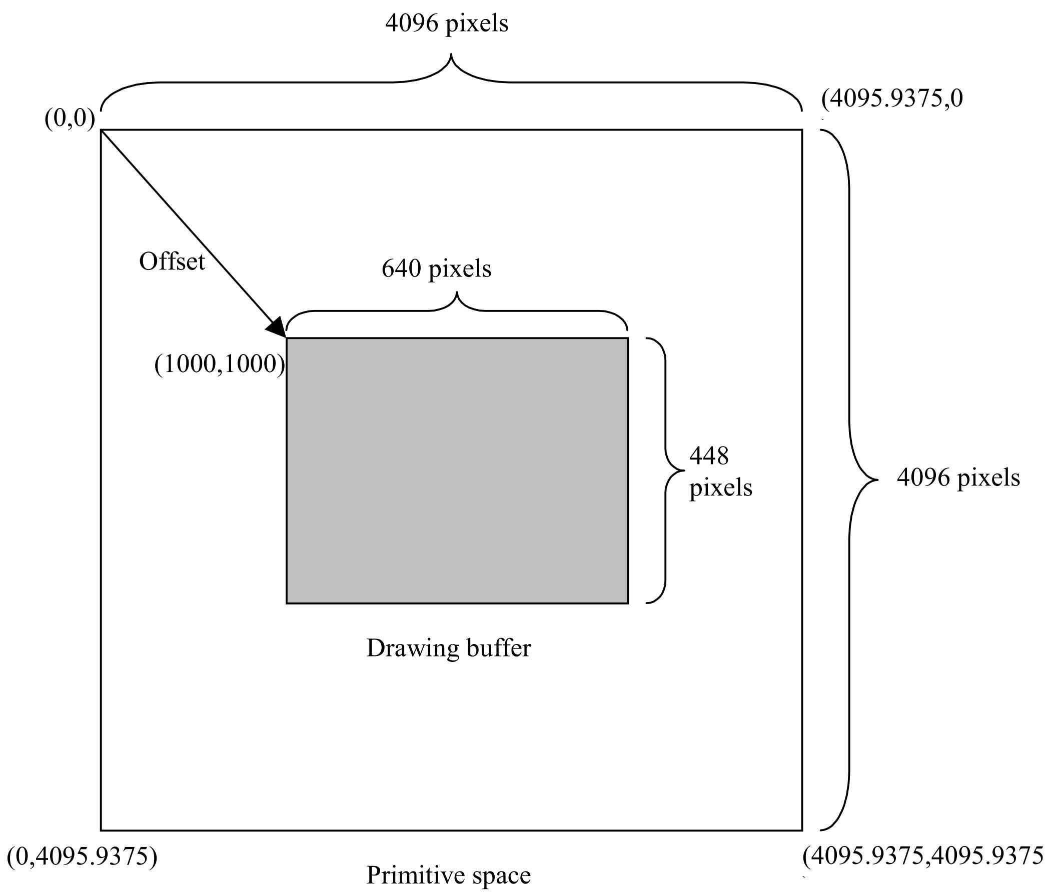
This is done by setting three GS parameters:

1. The width of the drawing buffer (i.e. the width of the screen, 640 pixels)
2. The height of the drawing buffer (i.e. the height of the screen, 448 pixels)
3. The top left corner in primitive space that the drawing buffer rectangle will be mapped to.

The setting of the last parameter is not obvious. At first glance, we would say that the top left corner should map to the top left corner of primitive space (0,0). But there is an important reason why we usually don’t want to do this.

When we are processing 3D graphics, it’s likely that some polygon vertices will transform to points that correspond to pixels off screen. This does not mean the polygon cannot be drawn, but it does create a problem if the top left co-ordinate of the screen is (0,0). If the top left co-ordinate is (0,0) and a polygon is transformed to a (X,Y) which is off the left and/or top of the screen, then its X and/or Y will be negative. But the GS cannot handle polygons where either X and/or Y lie outside primitive space, i.e. the range (0,4095.9375).

To get around this problem, the GS has two registers which are used as a drawing offset. After rasterising the polygon in primitive space, these offsets are *subtracted* from the numbers passed in as (X,Y), and the result is used to determine the correct part of the drawing buffer to write to.



In this diagram, we have set these offset register to be (1000, 1000). (This isn't what is used in the main program, by the way). To get polygons onto the screen, we must specify (X,Y) coordinates that are between (1000, 1000) and (1000 + 640, 1000 + 448).

But why is an offset of (1000, 1000) used in the first place? Wouldn't it be just as convenient to have an offset of (0,0)? Well, the advantage of having an offset at (1000, 1000) is that it is possible to pass in a polygon that is slightly off screen, such as (980,970) to (1200, 1300) to (1140, 1220). This triangle is partially on screen, but the first vertex is not in the range (1000, 1000) to (1000 + 640, 1000 + 448). However, the GS rasteriser is still able to draw it, because the triangle lies completely within primitive space. If the offset registers were set to (0,0) instead of (1000, 1000), then the top left co-ordinate of the polygon would be (-20, -30), which is outside primitive space and therefore cannot be rendered by the GS.

Since all polygons must lie within primitive space, the libraries set the default offset to map the drawing buffer directly into the middle of primitive space, so that the very centre of the drawing buffer is the very centre of primitive space. That is, if you specify 640 as your screen width, the X offset will be (4096 – 640)/2. If the Y is 448, then the Y offset will be (4096 – 448)/2.

For this program, the screen dimensions are 640x4448, so the X/Y offset is (1728, 1824). This means that primitive coordinates (1728.0,1824.0) correspond to the top left pixel of the screen, and (2367.0, 2272.0) corresponds to the bottom right pixel. The pixel in the middle of the screen is at (2048.0, 2048.0).

This explains why the X and Y of the polygons we are drawing in the program have coordinates relative to (2048.0, 2048.0). This is because this is the very centre of primitive space, and the default double buffer settings map our drawing buffer directly over the centre of primitive space.

The reason that the X/Y values are shifted left by 4 is to move the results into 12:4 fixed point format, as required by the GS.

Because the Z buffer is set to `SCE_GS_ZALWAYS` (The Z-Buffer is not updated), the Z value of the XYZ triple is irrelevant.

GS register queues

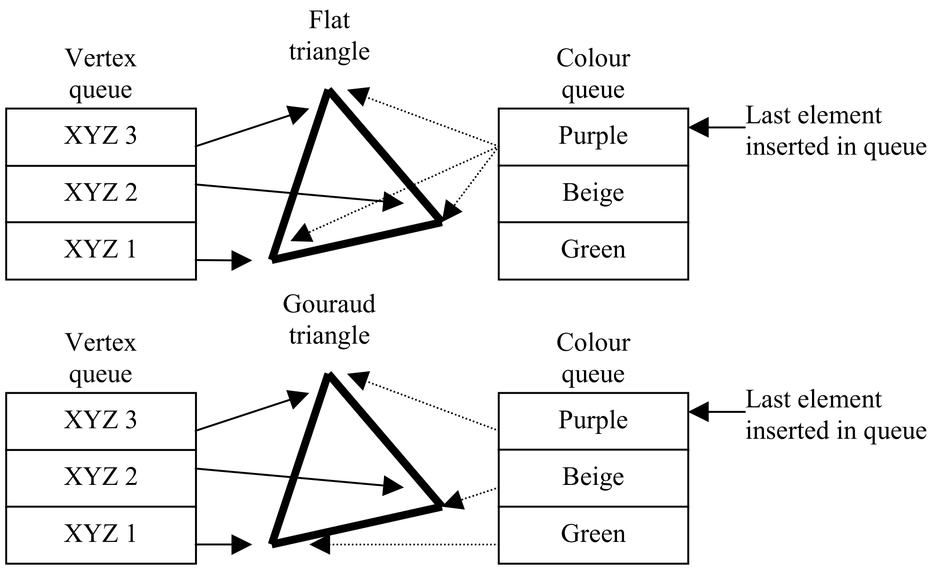
The GIF Tag we use sets the same XYZ register 3 times per triangle. You may think that setting the same register 3 times would simply overwrite the information that was already stored there. In general, this is correct. However, this does not happen for the XYZ register because the GS has an internal queue for this register. Setting the XYZ register multiple times adds the register setting onto the end of the XYZ queue.

There are 4 such register queues in the GS. These are the vertex queue, the colour queue, the STQ queue and the UV queue. The vertex queue holds XYZ coordinates for polygons to be drawn, the colour queue holds colours for each vertex, the STQ queue holds perspective correct texture coordinates, and the UV queue holds non-perspective correct texture coordinates.

The size of each queue depends on the type of primitive to be rendered. A triangle primitive has a queue 3 elements deep, because it has 3 vertices. A sprite or line has a queue 2 deep because they have two endpoints. A point's queue is 1 deep.

Every time you write to a register that has a queue, an appropriate value is removed and the queue is 'shunted' forwards before the new value is added to the end of the queue.

When drawing is activated, the GS uses the values in the registers and the register queues to render the polygon. For example, if a gouraud triangle drawing is activated, the GS will use the 3 values in the vertex queue as the 3 screen coordinates, and the 3 values in the colour queue as the vertex colours. If a flat triangle drawing is activated, the GS will use the 3 values in the vertex queue as the 3 screen coordinates, and the last value to be entered into the colour queue for the polygon colour.



The vertex queue is slightly different to the other queues. When the vertex queue is written to, you can optionally specify a *kick*. One register writes to the queue with a kick, and another register writes to the same queue without a kick. When a kick is specified *and the vertex queue is full*, then the GS starts drawing the type of polygon specified in the `PRIM` register.

In our program we write an XYZ coordinate to the 0x5 register 3 times, which is register `XYZ2 (kick)`. This register inserts the coordinates into the vertex queue *and* specifies a kick. The first two writes have no effect (the vertex is queued but the kick is ignored), because the vertex queue is not yet full. The GS has been told via the PRIM register that it will be drawing a triangle, so it is expecting 3 elements in the vertex queue. As it does not have enough information to draw a triangle, it does nothing.

On the third XYZ2 setting, the coordinates enter the vertex queue (filling it), and kick is specified. The GS can begin rendering, because kick has been specified and it has enough information to continue. Therefore after the 3rd write to XYZ, the GS will begin to draw a polygon.

```
*(spr++) = SCE_GS_SET_PRIM(
            SCE_GS_PRIM_TRI,    // PRIM (Primitive type)
            0,                  // IIP  (Gouraud)
            0,                  // TME  (Textured)
            0,                  // FGE  (Fogging)
            0,                  // ABE  (Alpha Blending)
            0,                  // AA1  (Anti-Aliasing)
            0,                  // FST  (Use ST for texture coords)
            0,                  // CTXT (Context)
            0);                 // FIX  (Fragment control)
*(spr++) = SCE_GS_SET_RGBAQ(0x00, 0x80, 0x00, 0x00, 0);
*(spr++) = SCE_GS_SET_XYZ((2048 - 40) << 4, ((2048 - 80) << 4), 0);
*(spr++) = SCE_GS_SET_XYZ((2048 + 40) << 4, ((2048 - 80) << 4), 0);
*(spr++) = SCE_GS_SET_XYZ((2048     ) << 4, ((2048 -160) << 4), 0);
```

This is data for the second triangle. Notice that the RGBAQ is different to the RGBAQ for the first triangle. This will make this triangle green.

By the time this data arrives at the GS, the vertex queue will be clear for two reasons.

1. When the `PRIM` register is set to draw a "TRI" (triangle) primitive, the GS will clear the vertex queue after it receives 3 vertices. Similarly, if PRIM was set to draw a "LINE" primitive, the GS will clear the queue after it receives 2 vertices.

2. Setting the `PRIM` register also clears the vertex queue.

This means that when the 4th XYZ co-ordinate is set (the first vertex of the second triangle), it will be the only element in the queue.

Sometimes we do not want the vertex queue cleared. This is very useful when drawing strips and fans. We will introduce triangle strips in a later example.

```
asm("sync.l");
```

The EE core has a small writeback buffer to speed up store instructions. Writes to the cache and memory are stalled until this internal writeback buffer is full. When the buffer is full, the contents are actually stored. The reason the writeback buffer exists is that it's faster for the EE core to make multiple writes at once than it is to write each store separately. However, this also means that the writeback buffer may be partially full when the program is ready to start the DMA transfer. Since the DMA does not know about the writeback buffer, you must flush the writeback buffer before starting the DMA to ensure that memory is consistent with what your program believes it to be.

The "`sync.l`" instruction flushes the writeback buffer.

```
    sceGsSyncV(0);
```

This call will wait for the vertical blank, the short period of time when the electron beam is making its way from the bottom back up to the top of the screen. This is the perfect time to swap double buffers.

```
    buffer = 1 - buffer;
    sceGsSwapDBuff(&db, buffer);
```

This call will swap the double buffers.

```
    sceDmaSendN(dmaGif, (u_long128*)(SPR | 0x80000000), 6);
```

This call will begin the DMA transfer to the GIF. Although this could be performed manually by setting the appropriate DMAC registers directly, this example uses the convenient SCE library function. The function call takes 3 parameters:

1. A pointer to the appropriate DMAC channel registers. This pointer was initialised by the previous `sceDmaGetChan` call. This program uses channel 2, which transfers data from memory or scratchpad to the GIF.
2. The start address of the DMA transfer. A slight peculiarity here is that when the transfer address is on scratchpad RAM, we must set the top bit of the address to 1. This is so the DMAC knows whether to access SPR or main memory. The DMAC ignores the other bits in the top nibble. The DMA transfer **must** be aligned on a quadword boundary.
3. The length of the transfer. The unit is always quadwords. In this case, the program transfers the GIF tag (1 quadword) and 2 triangles with 5 x 64-bits each (5 quadwords) for a total of 6 quadwords.

The program continually waits for the top of frame, swaps the double buffers and initiates the DMA transfer.

Summary

This completes our walkthrough of Example 1. This example:

- Set up double buffers and initialised the screen
- Built a GS primitive on scratchpad RAM
- Used the DMA to transfer it to the GIF.
- The GIF used the first quadword as the GIF tag. This lets the GIF decide how to interpret and where to send the data that is to follow.
- The result was 2 triangles on the screen, coloured red and green respectively.

In the next Example, we'll introduce DMA tags and cover the PACKED mode of the GIF tag.

Questions and problems

1. Q: What changes would have to be made to get 4 triangles drawn?

A: There would need to be 3 changes. Firstly the extra triangle data for the 2 new triangles would have to be generated and added onto the end of the DMA list. Secondly the GIF tag's `NLOOP` field would be updated to 4 instead of 2, since we need to loop though the `REGISTER LIST` 4 times. Finally the DMA length would have to be increased from 6 to 11, since we would now be DMA'ing 11 quadwords of data (1 GIF tag + 10 quadwords for the 4 triangles).

2. Q: What happens if the DMA quadword count is too long or too short?

A: If the DMA quadword count is too short, then not enough quadwords will be transferred and the GIF will be starved of data. If you re-send the DMA list (for the next frame), the GIF will be expecting the rest of the data from the first frame, but instead it will receive the original DMA list. It will interpret the incoming DMA list as the rest of the data, and hence will consider the first qword (GIF tag) to be data instead of a GIF tag. This will cause very strange results on screen.

If the DMA quadword count is too long, then extra quadwords following our DMA list will be transferred to the GIF. Once data of the correct length has been processed by the GIF, it will consider the first extra quadword to be a new GIF tag. Since this word is undefined, anything could happen.

Thus, it is important that your total DMA length reconciles with the total length of data specified in the GIF tag by the `NREG` and `NLOOP` and `FLG` fields. The number of quadwords expected to be in a GS primitive is calculated by the following formulae (if `NREG` is 0, then `NREG` is considered to be 16):

`FLG` = 0 (PACKED mode) : `NREG * NLOOP`
`FLG` = 1 (REGLIST mode) : `(NREG * NLOOP + 1) / 2`
`FLG` = 2 (IMAGE mode) : `NLOOP`.

If the GIF tag expects 10 quadwords, then the DMA will send 11 (1 for GIF Tag + 10 data). For new programmers, one of the most common mistakes to make is to change a GIF Tag's `NLOOP` or `NREG` without changing the length of the DMA transfer.

3. Q: What would happen if the X or Y coordinates are out of the range (0.0, 4095.9375)?

A: If the X and/or Y lie outside the range, then they are still interpreted as unsigned 12:4 fixed point format integers. This means that if you write the value –10.0 into the X field (-10.0 = 0xff60 in signed 1:11:4 format), this would be considered an X value of 0xff6.0, which is 4086.0 in 12:4 format. In this case, the X co-ordinate would be on the very far right hand side of primitive space, which is probably not what you wanted.

Section E - Example 2 – DMA tags and PACKED mode

In this example, we introduce DMA tags and the PACKED mode.

The DMA has two modes of operation. In *Normal Mode*, the DMA transfers a single contiguous area of memory. In *Chain Mode*, the DMAC reads special DMA tags stored in memory. These tags tell the DMAC the address and length of the data to transfer. These are used to transfer non-contiguous areas of memory to a peripheral.

A GIF tag can be set to PACKED mode. This is the most commonly used mode when creating dynamically creating GS packet data. It sets the GIF to interpret the quadwords as single GS register settings. It is convenient to use because the data generated by the VUs is designed to be processed in PACKED mode.

Example 2 - Walkthrough

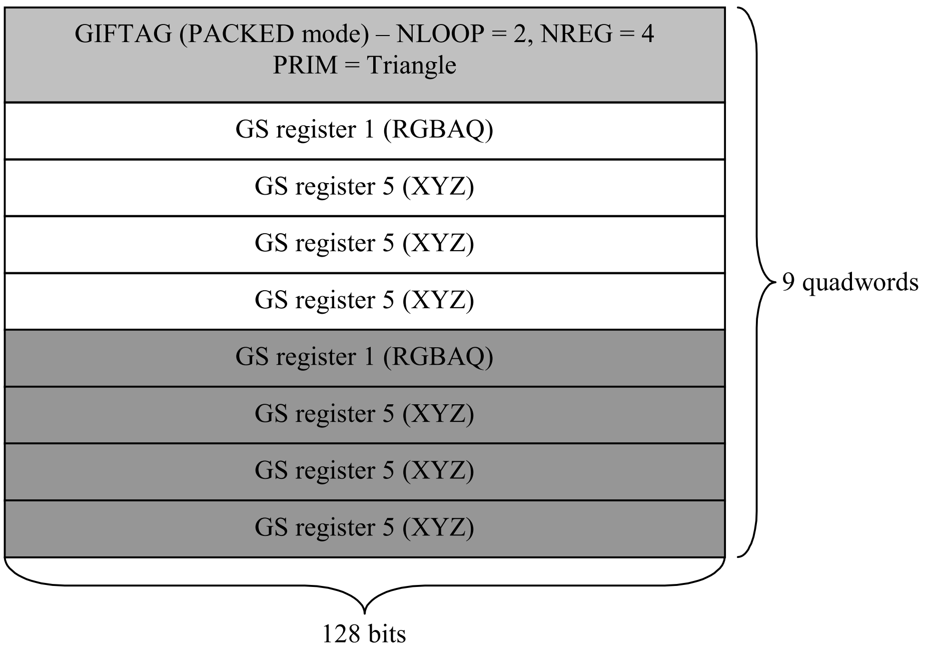
This program differs from the program above in only a few ways. We will not cover functions that are reused in the same way.

This program puts the same two triangles on the screen as before. In the previous method, the program created all the necessary data in one contiguous block on the scratchpad before using the DMA to send this block to the GIF. In this example however, the data is in various places around memory. We use *DMA tags* to build a list of memory areas to send to the GIF.

In the previous example, we used the GIF tag's REGLIST mode. In this mode, the GIF interprets each following quadword as two separate GS register settings. These two 64 bit values are written directly to the appropriate GS register.

In this example, we use PACKED mode. In PACKED mode, the GIF uses a single quadword to create a single GS register setting. This may sound inefficient (only 64 bits of the quadword will be used at most), but it is more convenient for reasons covered later.

The program sends the following data to the GIF:



This is different from example 1 in three ways:

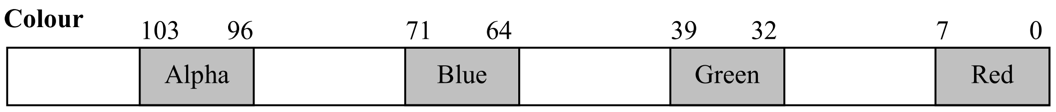
1. An entire quadword is sent for each GS register setting. The GIF will discard unused bits from each quadword according to its destination register, then pack the results into a 64 bit value it can use for a GS register setting.

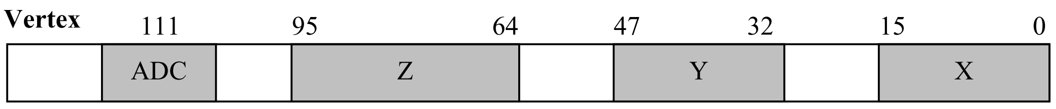
2. A `PRIM` GS register setting is no longer sent as part of the primitive data. Instead, the PRIM register is set by the GIF Tag using the PRIM and PRE fields.

3. The length of the transfer data is longer. It is 9 quadwords instead of 6.

```
u_long bluetriangle[8] __attribute__ ((aligned(16))) = {
    0x0000000000000000L, 0x0000000000000080L,               //Colour
    ((2048L - 40) << 4) | ( ((2048L + 40) << 4) << 32 ), 0,  //Vertex
    ((2048L + 40) << 4) | ( ((2048L + 40) << 4) << 32 ), 0,  //Vertex
    ((2048L     ) << 4) | ( ((2048L - 40) << 4) << 32 ), 0   //Vertex
};

u_long green[2] __attribute__ ((aligned(16))) = {
    0x0000008000000000L, 0x0000000000000000L                 //Colour
};
```

These two variables define quadwords that store colours and vertex data. The `bluetriangle` variable stores a colour, then 3 vertex settings. The `green` variable stores just a colour. These are stored in the following formats:





Only 32 out of the 128 bits in the colour quadword are used. Only 33 out of the 128 bits in the vertex quadword are used. The GIF will discard the unused bits (see below).

The arrays must be quadword aligned because the DMA can only transfer data that is quadword aligned. This is why `__attribute__ ((aligned(16)))` is at the end of each array definition. The `__attribute__ ((aligned(xx))` directive is a GNU C extension to the C language that asks the linker to align the variable on a specified byte boundary. In this case, we want quadword alignment (16 bytes).

```
const u_long myGIFTag[2] __attribute__ ((aligned(16))) = {
        SCE_GIF_SET_TAG(
            2,      // NLOOP
            1,      // EOP
            1,      // PRE
            SCE_GS_SET_PRIM(
                SCE_GS_PRIM_TRI,    // PRIM (Primitive type)
                0,                  // IIP  (Gouraud)
                0,                  // TME  (Textured)
                0,                  // FGE  (Fogging)
                0,                  // ABE  (Alpha Blending)
                0,                  // AA1  (Anti-Aliasing)
                0,                  // FST  (Use ST for texture coords)
                0,                  // CTXT (Context)
                0),                 // FIX  (Fragment control)
            0,      // FLG
            4       // NREG
        ),
        0x0000000000005551L
    };
```

This is the GIF tag definition. In the original example, one of the fields in the `REGISTER LIST` wrote to the `PRIM` register. As it happens, the only significant bits of the `PRIM` register are the lower 10 bits. The

GIF designers had more than that spare within the GIF tag, so they allowed the PRIM register to be set from within the GIF tag.

To do this, set the PRIM field in the GIF tag to the desired primitive type and attributes, and set the PRE bit to 1. The PRE bit activates the PRIM field within the GIF tag. The GIF will then use the PRIM field to set the PRIM register. The PRIM register is only set once, when the GIF tag is read.

There is one proviso – the PRE/PRIM bits are only valid when the GIF tag is set to PACKED mode (FLG=0). The PRE/PRIM bits have no effect in REGLIST and IMAGE mode.

DMA tags

In example 1, the data was transferred was in one continuous section. However, when we send data to a peripheral it may be in many pieces scattered around memory.

For example, say you want to transfer a VU microprogram to process and render a model that has a texture. You need to transfer the microprogram, the model data and the texture to the VIF peripheral, but these are likely to be placed in different areas of memory instead of a single contiguous block. With older architectures, the only solutions to this problem would be to either copy all the data into a contiguous block or initiate multiple DMA transfers.

On the PlayStation 2, you have two options. Either a single area of RAM can be transferred (as performed in Example 1), or an instruction list for the DMA can be built, which will tell it which areas of RAM to transfer. These instructions are 64 bits wide and are called *DMA tags*.

There are 8 different types of tags. Each type of tag tells the DMAC:

1. Where to get the data from.
2. How much data there is to transfer.
3. Once the data has been transferred, where to find the next tag.

The DMAC will read a DMA tag, transfer the data that the tag refers to, then read the next tag, transfer that tag's data, and so on. The DMA transfer ends after certain tag types are processed.

An important note: **Although DMA Tags are 64-bits wide, they must always be quadword aligned.**

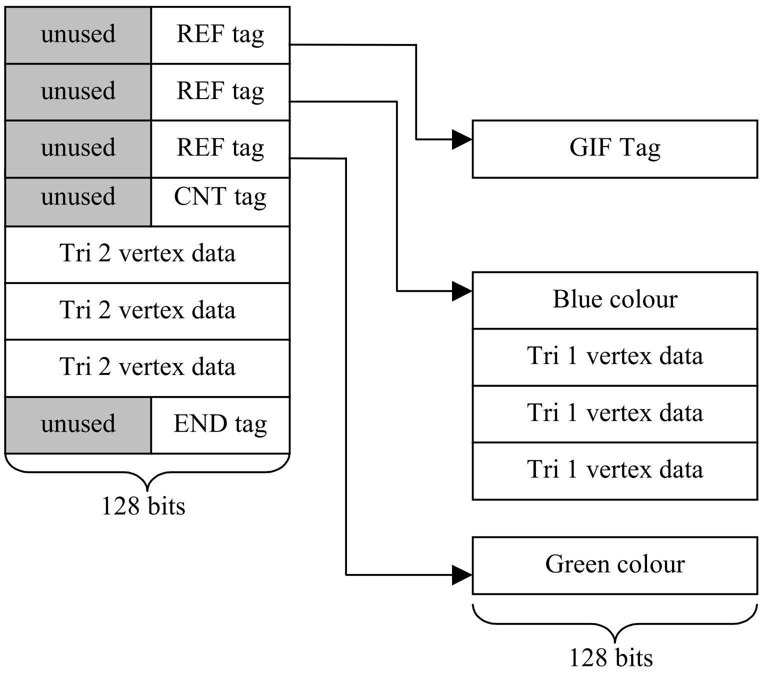
We will use 3 types of tags in this example.

1. The REF tag contains an address, and the number of quadwords to transfer from that address. When this is complete, the DMAC will examine the quadword after the REF tag for the next DMA tag.

2. The CNT tag only specifies the number of quadwords to transfer. These are transferred starting from the quadword address directly following the CNT tag. When the transfer is complete, the DMAC obtains the next DMA tag from the quadword following the last quadword transferred.

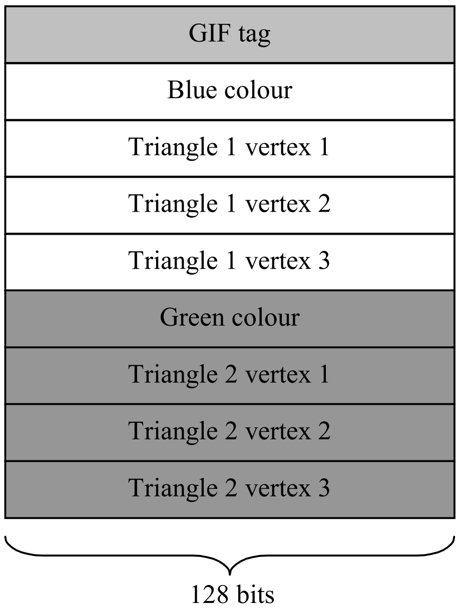
3. The END tag is the same as the CNT tag, except that when the transfer is complete, the DMA transfer is halted instead of continuing to the next DMA tag.

The program uses these tags to reference the data contained in the GS packet. Firstly it uses a REF tag to send the GIF tag. This is followed by a REF that sends the colour and vertex data for triangle 1. Another REF is used to send the colour for the second triangle, and a CNT is used for the vertex data of the second triangle. Finally, an END tag is used to halt the DMA transfer. Our END tag transfers no data.



The data is organised in memory a little strangely (you'd probably never see this in 'real' code), but this is just for example purposes.

The DMAC will parse the DMA tags and send the GIF a stream of quadwords. The GIF will receive the following quadwords:



The GIF knows nothing of the memory locations of the source data, it only receives a stream of quadwords from the DMA.

A DMA tag has the following structure:

DMA Tag

63 32	31	30 28	27 26		14 0
ADDR	IRQ	ID	PCE	unused	QWC

- The QWC field is the QuadWord Count. It holds how many quadwords will be transferred by this tag. It is 15 bits in size, implying a maximum QWC of 0x7fff. This is equivalent to just under 1MB of data.
- The PCE field is the Priority Control Enable field. We won’t be using priority control in this example.
- The ID field is a 3 bit field that defines the type of tag. The setting of the ID field affects how the ADDR field is interpreted.
- The IRQ field is the Interrupt ReQuest field. When this is set, an interrupt is generated after the tag has completed transferring it’s data. The example program does not use this feature.
- The ADDR field is an address. The way the address is used depends on the ID.

For a full description of all the different types of DMA tag, see Section 3.6 in the EE Users Manual.

When the REF tag is specified in the ID field, the ADDR field is interpreted as the starting address of the data to transfer. QWC is the number of quadwords to transfer from that address. The DMAC uses the quadword following the REF DMA tag as the next DMA tag.

When CNT or END is specified, the ADDR field is not used.

```
        // Create REF DMA Tag for GIFTag
    addr = (u_long)&myGIFTag;
    *(spr++) = (addr << 32) | (3L << 28) | 1L;       // REF DMA tag
    *(spr++) = 0;      // padding
```

These lines create a REF DMA tag. The first line gets the address of the area to be transferred. The second writes the entire tag. The `(3L << 28)` sets the ID to REF. The `1L` sets the quadword count, since the tag will transfer 1 quadword (the GIF tag).

The last line pads the unused 64 bits following the GIF tag. We’ll see in a later example how these bits can be useful when dealing with the VIF. But in this case, the transfer is directly to the GIF. Since the DMA data and the DMA tag must be quadword aligned, this 64-bit word is unused.

```
            // Triangle 1

        // Create REF DMA Tag for blue triangle
    addr = (u_long)&bluetriangle;
    *(spr++) = (addr << 32) | (3L << 28) | 4L;       // REF DMA tag
    *(spr++) = 0;      // padding
```

This is another REF tag. It is placed directly after the first REF tag. When the transfer for the first REF tag is complete, the DMAC will examine the quadword following it for the next tag. It will contain this REF tag.

This REF tag points to the blue triangle’s colour and vertex data. This is 4 quadwords of data to transfer, so the QWC is set to 4.

```
            // Triangle 2

        // Create REF DMA Tag for green colour
    addr = (u_long)&green;
    *(spr++) = (addr << 32) | (3L << 28) | 1L;       // REF DMA tag
    *(spr++) = 0;      // padding
```

So far the program has built a DMA instruction list that transfers a single triangle. Now it will add the colour and vertex data for the second triangle. The REF tag above refers to the green colour that was defined previously. It transfers a single quadword, the colour of the new triangle.

```
        // Create CNT DMA Tag for triangle vertex data
```

```
    *(spr++) = (1L << 28) | 3L;                      // CNT DMA tag
    *(spr++) = 0;      // padding

    *(spr++) = ((2048L - 40) << 4) | ( ((2048L - 80) << 4) << 32 ); // X and Y
    *(spr++) = 0;      // Z
    *(spr++) = ((2048L + 40) << 4) | ( ((2048L - 80) << 4) << 32 ); // X and Y
    *(spr++) = 0;      // Z
    *(spr++) = ((2048L     ) << 4) | ( ((2048L -160) << 4) << 32 ); // X and Y
    *(spr++) = 0;      // Z
```

Now a new type of tag is used, the `CNT` tag. `CNT` is short for "CoNTinue", as it continues reading the quadword after the transfer data to get the next tag. This tag needs only specify the QWC. It uses the quadword address following the tag as the start address of the transfer. QWC is 3, so it will transfer the 3 quadwords following the `CNT` tag. This happens to be the triangle vertex data (one XYZ per quadword, in the format described above).

```
        // Create END DMA Tag for triangle vertex data
    *(spr++) = (7L << 28) | 0L;                      // END DMA tag
    *(spr++) = 0;      // padding
```

This is the END tag that will end the transfer. Although it can optionally transfer data just like CNT, the QWC is set to 0, so it will not transfer any data. Any type of DMA tag can have a QWC of 0.

```
        sceDmaSend(dmaGif, (u_long128*)(SPR | 0x80000000));
```

`sceDmaSendN` (used in Example 1) will set the DMAC channel to *Normal Mode*, where a single contiguous area of memory is transferred. The transfer area is considered to be purely transfer data. `sceDmaSend` (used in this method) sets the DMAC channel to *Chain Mode*, where the DMAC interprets the address contents as DMA tags and data. It is not necessary to pass a data length to `sceDmaSend`, because the quadword counts are contained within the DMA tags.

What the GIF receives

The GIF knows that the DMA will be sending it 128-bit quadwords, but it does not know about how the DMA works. It makes no difference to the GIF where the data is coming from, whether it be the SPR or main memory, in one continuous lump (Example 1) or broken up into many different pieces (as in this example). All the GIF receives is the stream of quadwords sent to it by the DMAC. From the GIF's point of view, it is just receiving a continuous stream of data from the DMA over PATH3.

This is true for all peripherals that the DMAC can send data to. They all receive a stream of quadwords from the DMAC. They get no information about how the data is actually organised in memory.

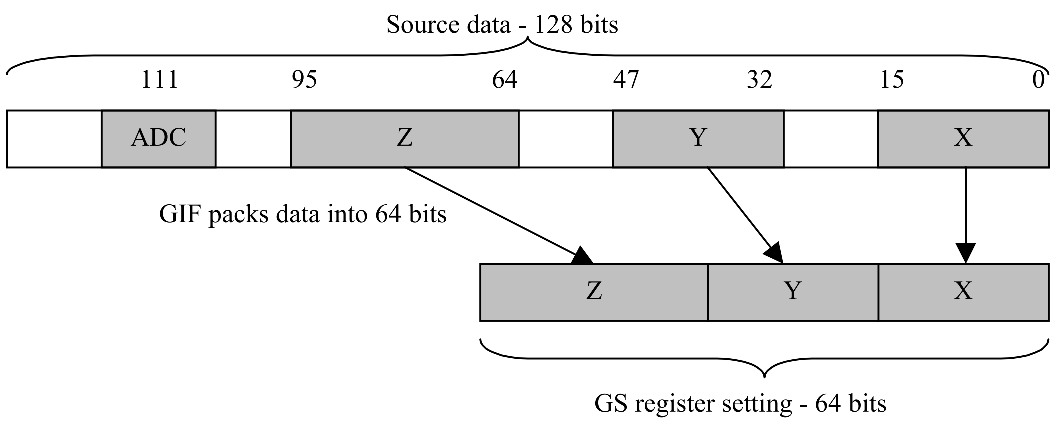
GIF PACKED mode

The GIF receives a stream of quadwords from the DMA. These include the GIF Tag, the colour settings and the vertex settings. When the GIF tag specifies PACKED mode, it considers all quadwords that follow to contain one GS register setting. The two formats used in this program are described above. Each GS register is 64 bits wide, so the GIF must pack each quadword into 64 bits.

It does this depending on the GS register that will be written to (specified by the corresponding field in the `REGISTER LIST`).

For example, say that the current `REGISTER LIST` field was 5, which is the XYZ register.

The ADC field is a one bit field. When it is 0, the GIF writes to GS register 5 (XYZ with kick). When it is 1, the GIF writes to GS register 0xD (XYZ without kick).



The GIF reads the quadword and discards the unused bits. Whatever remains is packed into 64 bits and used as the GS register setting.

A full description of all the quadword formats and the way the GIF packs them into 64 bit fields is contained in Section 5.3.2 of the EE Users Manual.

Why use PACKED mode?

It may seem a bit strange to organise data this way. Why would we store a GS register setting as a quadword when it would be much more memory efficient to store it in 64 bits (as we did in Example 1)? The reason is because the VUs work in units of 128 bits, and 128-bit values are much easier to work with when the VUs are dynamically creating GS register settings.

For example, if you performed a transform calculation on a vertex, you would end up with a 128-bit vector [X, Y, Z, W] in a 128-bit VU register. If these values were converted to integers and the register was saved to memory, then the 128-bit value in memory would look very much like the unpacked format above (except the ADC bit would not be set properly). It is then possible to DMA the data straight to the GIF, which will pack it up into 64 bits. This saves the VU some hard work reorganising the data into 64 bits.

We will use PACKED mode for the rest of the samples. It is the most commonly used mode for dynamically generated data. REGLIST mode is more appropriate for pre-calculated static data, as it uses less space.

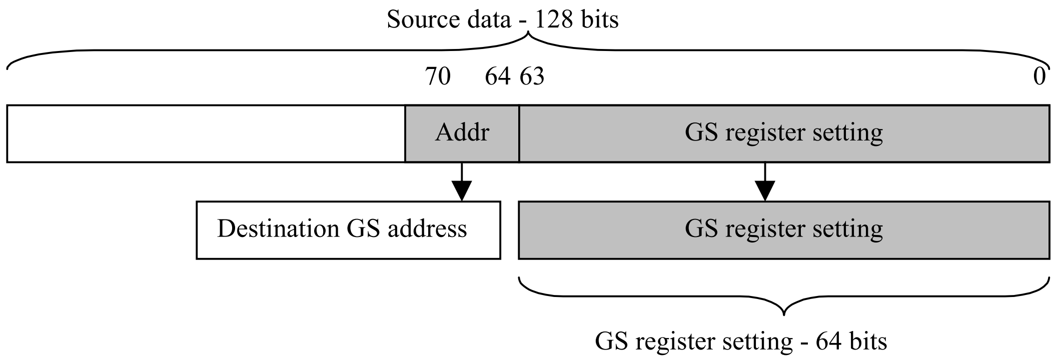
More about the REGISTER LIST

The `REGISTER LIST` in the GIF Tag has fields that are only 4 bits wide. This means special conventions must be used to address the full 7 bit address range inside the GS.

For the most part, the 4 bit number specified in the `REGISTER LIST` is considered to be the destination GS address. The most commonly used GS registers (primitive type, colour, vertex and texture settings) all have GS register addresses less than 0x10, so their addresses all fit into 4 bits.

To access the rest of the address range, it is necessary to use an indirect method. This is called A+D mode, and is only available when using PACKED mode. It is not available in any other mode.

When the current `REGISTER LIST` field is set to 0xE, the GIF interprets the quadword as follows:



This allows you to write to any address in the 7-bit GS register address range.

Summary

In this example, we split the GS packet up so that it was placed in various areas of memory. Then we built a DMA list using DMA tags that would transfer these areas to the GIF.

The GS packed data was in PACKED mode format, where each GS register setting is a single quadword in size, and the GIF packs it into 64 bits.

The next example covers 3D calculations and uses triangle strips.

Questions and problems

1. Problem: Without making any other changes, remove the last END tag and replace the CNT tag with an appropriate DMA tag that will transfer the second triangle's vertices *and* end the transfer.

Solution: Replace the `CNT` tag with an `END` tag. An `END` tag will transfer the data just like a `CNT` tag, but will end the DMA transfer directly afterwards.

2. Q: The `PRIM` register is only set once, and two triangles are drawn. Since the vertex queue will be full after the first triangle is drawn, why are triangles not drawn when the 4th and 5th vertices are sent?

A: Because the primitive type is set to Triangle (not Tri-strip or Tri-fan), the vertex queue is cleared after 3 vertices have been received. If the primitive type were tri-strip or tri-fan, then triangles would be drawn when the 3rd, 4th, 5th and 6th vertices are sent.

3. Problem: Change the primitive type in the GIF tag to a tri-strip (`SCE_GS_PRIM_TRISTRIP`). If you re-run the program, there will be extra triangles drawn. Make a change to the data you are sending to prevent this.

Solution: The extra triangles are being drawn because when the 4th and 5th vertices are added, the vertex queue is full and the vertices are written to the 'kick' register. Set the `ADC` bit (bit 111) in the vertex data quadword for the 4th and 5th registers, which will cause the GIF to send the vertices to the vertex queue without a kick. (This means changing 2 lines from "`*(spr++) = 0;`" to "`*(spr++) = 1L << 47;`")

Section F - Example 3 – VU macromode

In this example, the VU0 macromode library and triangle strips are introduced.

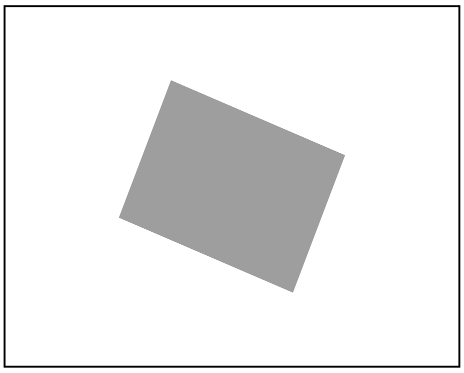
Most applications on the PlayStation 2 will have some 3D element to them. The best way to perform 3D calculations is using matrices and vectors, and the Vector Units are designed to process these at high speed. The VU0 macromode is introduced here to give an idea of how the VUs are designed and how they perform 3D calculations.

This example creates a very simple 3D model (a single square), and rotates it in 3D around 3 axes.

The VU0 macromode library (`libvu0`) is a SCE library that uses the VU0 for calculations. The source to these libraries is provided in `/usr/local/sce/ee/src/lib/vu0`. Some of the functions in the source are examined later in this section to help you get a feel for how the VU0 performs calculations.

How the program works

This program draws a quadrilateral on screen, and rotates it in 3D. No lighting or texturing is performed.


Program output

We use homogenous coordinates and 4x4 matrices for our vertices and 3D transformations. We won't go into a discussion of homogenous coordinates here (they are covered in such books as "Computer Graphics" by Foley/Van Damme/et al), but briefly, homogenous coordinates use 4x4 matrices because these can contain any type of transformation (rotation, translation, scaling and perspective). The 4x1 vectors used for the vertices are the standard 3x1 [X, Y, Z] triples, with an extra value 'W' appended to the end. For vertices, this value should always be 1.0 (i.e. [X, Y, Z, 1.0]).

The PlayStation 2 hardware is designed to deal primarily with homogenous coordinates. It can multiply a 4x1 vector by a 4x4 matrix with 4 instructions, and can scale a 4x1 vector with a single instruction.

Three matrices transform the vertices. These are LocalToWorld (Local coordinates to World Coordinates), followed by WorldToCamera (World coordinates to Camera coordinates), followed by CameraToScreen (the perspective transformation).

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ W' \end{bmatrix} \Leftarrow \begin{pmatrix} \text{Camera} \\ \text{to} \\ \text{Screen} \\ \text{matrix} \end{pmatrix} \times \begin{pmatrix} \text{World} \\ \text{to} \\ \text{Camera} \\ \text{matrix} \end{pmatrix} \times \begin{pmatrix} \text{Local} \\ \text{to} \\ \text{World} \\ \text{matrix} \end{pmatrix} \times \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

These matrices are multiplied together so the vertex vector only needs to be applied to one matrix. (We also see that in this system, transformations are applied from right to left – the rightmost transformation happens first. If they were to be left to right, then all the matrices would need to be transposed).

$$\begin{pmatrix}\text{Local}\\ \text{to}\\ \text{Screen}\\ \text{matrix}\end{pmatrix} \Leftarrow \begin{pmatrix}\text{Camera}\\ \text{to}\\ \text{Screen}\\ \text{matrix}\end{pmatrix} \times \begin{pmatrix}\text{World}\\ \text{to}\\ \text{Camera}\\ \text{matrix}\end{pmatrix} \times \begin{pmatrix}\text{Local}\\ \text{to}\\ \text{World}\\ \text{matrix}\end{pmatrix}$$

$$\begin{pmatrix}X'\\ Y'\\ Z'\\ W'\end{pmatrix} \Leftarrow \begin{pmatrix}\text{Local}\\ \text{to}\\ \text{Screen}\\ \text{matrix}\end{pmatrix} \times \begin{pmatrix}X\\ Y\\ Z\\ W\end{pmatrix}$$

Once the original vertex is multiplied by the transformation matrix (LocalToScreen), the resulting (X', Y', Z', W') vector needs to be converted from homogenous coordinates to standard (X, Y, Z) coordinates. This is performed by dividing each element by W.

$$\begin{pmatrix}\text{Screen X}\\ \text{Screen Y}\\ \text{Screen Z}\\ 1.0\end{pmatrix} \Leftarrow \begin{pmatrix}X'/W'\\ Y'/W'\\ Z'/W'\\ W'/W'\end{pmatrix}$$

However rather than perform 4 divides (the VUs only have one divide unit each), it is more efficient to calculate 1/W' and multiply the vector by that.

$$Q \Leftarrow 1/W'$$

$$\begin{pmatrix}\text{Screen X}\\ \text{Screen Y}\\ \text{Screen Z}\\ 1.0\end{pmatrix} \Leftarrow Q \times \begin{pmatrix}X'\\ Y'\\ Z'\\ W'\end{pmatrix}$$

So the process of calculating a 3D screen co-ordinate for a vertex is:

1. Multiply the vertex by the 4x4 transform matrix.
2. Calculate Q = 1/W'. (W' is the transformed W).
3. Multiply the vector by Q. Then X, Y, and Z will contain the screen coordinates and Z buffer depth.

The program calls functions in the VU0 macromode library that assist these calculations. The library can move data to and from VU registers and call VU0 instructions directly from the EE core. This can only be performed when the VU0 is not running a microprogram. Using the VU0 in this way is called "macromode". When the VU0 or VU1 run microprograms, this is called "micromode". Only the VU0 can be used in macromode.

C program walkthrough

```
#include <libvu0.h>
```

This is the header file for the VU macromode libraries.

```
#define PI (3.141592f)
```

We define π here. Notice the '`f`' at the end of the number. This tells the compiler that the number should be considered to be single precision floating point (32-bit) instead of the default, which is double precision (64-bit). This is very important, because all the floating point units on the PS2 are single precision. If a double precision number is encountered, the compiler will use software processing to handle it rather than the built-in FPUs, and the software processing is very very slow. All numbers should be defined to be floats rather than doubles.

```
#define SETVECTOR(v,x,y,z,w)  ((v)[0]=x,(v)[1]=y,(v)[2]=z,(v)[3]=w)
```

This is a macro used to set a 4-vector. The VU0 libraries only operate on 4-vectors and 4x4 matrices.

```
sceVu0FVECTOR vertices[4] = {       // vertices of the square
    { -100.0f, -100.0f, 0.0f, 1.0f },
    {  100.0f, -100.0f, 0.0f, 1.0f },
    { -100.0f,  100.0f, 0.0f, 1.0f },
    {  100.0f,  100.0f, 0.0f, 1.0f }
};
```

This defines the XYZ vertex positions of a square plane. Notice the W value is set to 1.0. When working with homogenous coordinates, it is necessary for the vertex to have 1.0 in this field, otherwise the transformation will be incorrect.

```
u_long polyColour[2] __attribute__ ((aligned(16))) = {
    0x0000008000000000L, 0x0000000000000000L
};
```

The colour of the polygon to be drawn is defined here. This is in the same quadword format used in Example 2, as the program uses PACKED mode to upload it, as before.

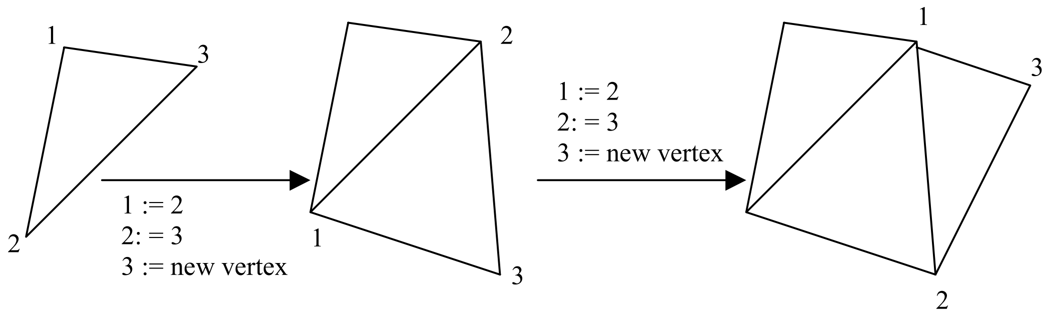
Triangle strips and fans

Below is the GIF Tag used. It is very similar to the GIF Tag used in example 2, except that it expects 4 vertices instead of 3, and NLOOP is 1.

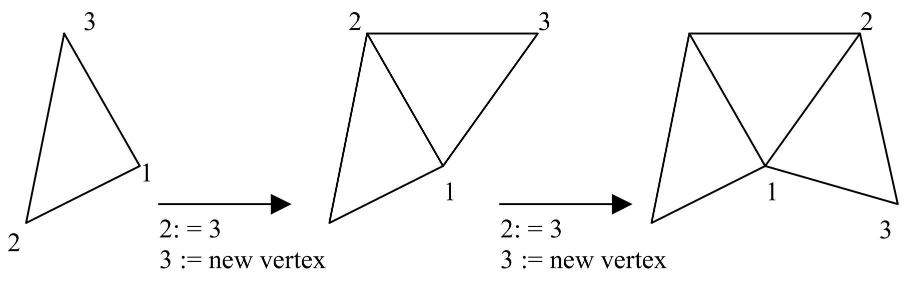
```
const u_long squareGIFTag[2] = {
    SCE_GIF_SET_TAG(
        1,          // NLOOP (Leave at 0, libraries fill this in)
        1,          // EOP (End Of Packet)
        1,          // PRE (PRIM field Enable)
        SCE_GS_SET_PRIM(    // PRIM field if PRE = 1
            SCE_GS_PRIM_TRISTRIP,   // PRIM (Type of drawing primitive)
            0,                      // IIP (Flat Shading/Gouraud Shading)
            0,                      // TME (Texture Mapping Enabled)
            0,                      // FGE (Fogging Enabled)
            0,                      // ABE (Alpha Blending Enabled)
            0,                      // AA1 (1 Pass Anti-Aliasing Enabled)
            0,                      // FST (Method of specifying Texture Coords)
            0,                      // CTXT (Context)
            0                       // FIX (Fragment Value Control)
            ),
        0,          // FLG (Data format, 0 = PACKED, 1 = REGLIST, 2 = IMAGE)
        5),         // NREG (Number of registers)
    0x55551L
};
```

When drawing polygon models, it is common for adjacent polygons to share vertices. For the sake of optimisation, it is best if the adjacent polygons are arranged into strips and fans. This is because after the initial 3 vertices have been sent, only one more vertex has to be sent to draw the next polygon in the strip/fan.

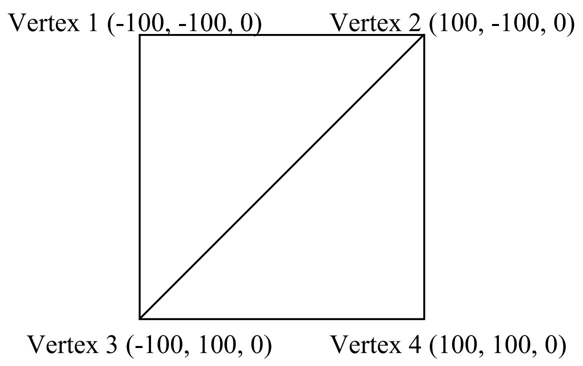
In a triangle strip, the last two vertices from the previous triangle become the first two vertices of the next triangle.



In a triangle fan, the first vertex is always part of the new triangle:



We will use a triangle strip to draw a quadrilateral. This quadrilateral is comprised of two triangles, but because they are adjacent, we only have to send 4 vertices instead of 6:



When the first 3 vertices are sent, the top left triangle is drawn. Upon sending the 4th vertex, the GS will reuse vertices 2 and 3, and draws the bottom right triangle.

```
sceVu0FMATRIX                    // Vertex transformation matrices
    LocalToWorldMatrix,
    LocalToScreenMatrix,
    WorldToScreenMatrix,
    CameraToScreenMatrix,
    WorldToCameraMatrix;

sceVu0FVECTOR rotation;          // Holds the vector that is used to create
                                 // the rotation matrix
sceVu0FVECTOR translation;       // The translation portion of the rotation matrix
```

The quadrilateral to be drawn will rotate in 3D space, then be projected using the standard homogenous co-ordinate matrix multiplication sequence:

```
Screen (X,Y,Z,W) = CameraToScreen * WorldToCamera * LocalToWorld * Vertex (X,Y,Z,W)
```

(where CameraToScreen, WorldToCamera and LocalToWorld are all 4x4 transformation matrices).

The variables above store the corresponding matrices, and the result of resulting one matrix by another. The rotation vector stores an Euler vector (X, Y, Z) holding the axis angles around which to rotate the object. The object will be rotates by Z first, then Y, then X. The translation vector moves the object in LocalToWorld space.

The VU0 macromode library uses some unusual types to refer to its elements. A `sceVu0FVECTOR` structure is actually typedefed to an array[4] of `float`. A `sceVu0FMATRIX` is typedefed to an array[4][4] of `float`. If a variable has been defined with type `sceVu0FVECTOR`, you can refer to an element in that vector using array offsets, e.g. `myVector[2]` will access the Z component.

```
    float delta_rot = 1.0f * PI / 180.0f;
```

This is the rotation speed of the object.

```
            // Create Local to World matrix
        {
```

```
        sceVu0FMATRIX work;
        sceVu0UnitMatrix(work);
        sceVu0RotMatrix(work, work, rotation);
        sceVu0TransMatrix(LocalToWorldMatrix, work, translation);
    }
```

The rotation and translation matrix is created here. This is put into the LocalToWorld matrix.

```
            // Create the final Local->Screen matrix
    sceVu0MulMatrix(LocalToScreenMatrix, WorldToScreenMatrix, LocalToWorldMatrix);
```

The LocalToWorld matrix is multiplied by the WorldToScreen matrix, which results in the final LocalToScreen transformation matrix.

```
    spr = (u_long *)SPR;
    *(spr++) = squareGIFTag[0];
    *(spr++) = squareGIFTag[1];
    *(spr++) = polyColour[0];
    *(spr++) = polyColour[1];
```

The DMA list is begun on the scratchpad. The program copies the GIF tag and the colour. These are each a quadword in size.

```
        sceVu0ApplyMatrix(V, LocalToScreenMatrix, vertices[i]);
        q = 1.0f / V[3];                // q = 1/V.w

        for(j = 0; j < 4; j++)
            V[j] *= q;
```

Once the LocalToScreen matrix has been set up, it is multiplied by each vertex. This will give us the final screen coordinates in homogenous coordinates (X,Y,Z,W). To convert this back to normal coordinates (X,Y,Z) we need to divide each element by W. Rather than perform 4 divide operations (very slow), we multiply each element by 1/W.

```
    V[3] = 0.0f;                        // Clear the 'W' value in XYZW.
```

The W value is now irrelevant. However, the ADC bit lies within the space held by W. To suppress a kick, this field needs to be changed so that bit 111 is set. However, in this example it makes no difference if ADC is not set because no drawing operation will occur when the first two vertices are sent. This is because the vertex queue will not yet be full.

```
        sceVu0FTOI4Vector((void *)spr, V);      // Convert and save V
```

This converts the floating point vector back to integers and stores the resulting quadword at the address `spr`. This format is conveniently what is required for XYZ in PACKED mode. Recall that the GS requires the format of the integers to be in unsigned fixed point 12:4 format. The VUs have an instruction that performs this conversion, and this instruction is used by `sceVu0FTOI4Vector`. This will convert the X/Y numbers to the correct fixed point format.

```
sceVu0ViewScreenMatrix(
    CameraToScreenMatrix,
    512.0f,
    1.0f,
    1.0f,
    2048.0f,
    2048.0f,
    1.0f,
    16777215.0f,
    64.0f,
    65536.0f
    );
```

This function is called in `init()`. It creates a matrix used to transform the Camera view into Screen space. The parameters specify:

1. Destination matrix (CameraToScreenMatrix)
2. Distance to the screen (512.0). This can be considered the standard Field Of View value. A lower value means a wider field of view.

3. Scaling factor for X/Y (1.0, 1.0)
4. Screen centre X/Y coordinates. These should correspond to the centre of the screen, and should be the same as the primitive space pixel value for the centre of the screen. (2048.0, 2048.0). These values are used to transform any 3D coordinates so that their origin is centred around the (X,Y) specified. This is so you do not have to manually add the offset to move them to the correct position in GS primitive space.
5. Minimum and maximum allowable Z values for this Z buffer. This is (1.0, 16777215.0) for a 24 bit Z-buffer.
6. Nearest and furthest Z coordinates that the application expects to use. (64.0, 65536.0). These two values and the min/max allowable Z values for the Z buffer are used to scale the transformed Z so that it uses the full breadth of the Z buffer rather than just a small range. These values are usually obtained experimentally.

A closer look at the VU0 library

Fortunately the VU0 macromode library source is provided. This is stored in `/usr/local/sce/ee/src/lib/vu0`. It is worthwhile examining some of the functions in this library to get an idea of how the VU0 works.

The VU0 macromode library uses EE assembly language instructions to control the VU0. This can only be performed when the VU0 is not running any programs.

In a previous chapter, we mentioned that the VUs have a VLIW architecture, where two instructions (one for the floating point units, one for the integer units) can be executed simultaneously. In macromode however, *either* the floating point units can be used, *or* the integer unit used at one time. It is possible to mix floating point and integer instructions in macromode, but only one unit can be used at once. Of course, this restriction does not apply to VU micromode programs.

Add vector

Although we didn't use it in our example, we'll examine this function first, as it's one of the simplest.

```
void sceVu0AddVector(sceVu0FVECTOR v0, sceVu0FVECTOR v1, sceVu0FVECTOR v2) {
    asm __volatile__("
        lqc2    vf4,0x0(%1)
        lqc2    vf5,0x0(%2)
        vadd.xyzw      vf6,vf4,vf5
        sqc2    vf6,0x0(%0)
    ": : "r" (v0) , "r" (v1), "r" (v2));
}
```

This function takes pointers to two input vectors (`v1` and `v2`), adds them together and saves the result to vector v0. It calls assembly language opcodes directly from the C program, so these are enclosed in an '`asm`' statement. The complex-looking last line is simply information to tell the compiler about the register usage of the assembly language section.

```
        lqc2    vf4,0x0(%1)
```

`lqc2` means "Load Quadword into Co-processor 2". Co-processor 2 is the VU0. `vf4` is the floating point vector register number 4 in the VU0, commonly called VF04. `v1` is a pointer to the vector, and this is referred to by the C compiler as '`%1`'.. Therefore this instruction says "Loads the 128 bit vector at address (0x0 + `v1`) into VU0 floating point register VF04.

There are 32 floating point vector registers in the VU0. Each register holds 4 single precision floating point numbers. Each floating point number is referred to by its field name (X, Y, Z and W). All registers are free for use. VF00 is special - it is always set to (0, 0, 0, 1), even if you store to it.

```
        vadd.xyzw      vf6,vf4,vf5
```

The two `lqc2` instructions load the vectors to be added. The instruction above performs addition. Most floating point instructions have 1 or 2 register sources and an arbitrary register destination. The destination is always the left-most argument, so here the destination is register VF06.

Field mask

The opcode itself has the suffix "`.xyzw`", called a *field mask*. This suffix indicates which fields are to be added (in this case, all fields). All vector floating point instructions can use such a suffix. If a field is not present in the suffix, then the result is not written for that field.

For example, if the suffix was "`.xz`", then only the X and Z fields would have been added together. The Y and W fields would have remained untouched. The only values to change would be VF06.X and VF06.Z. Whatever was in VF06.Y and VF06.Z before the add would still be there after the add.

In this example, the suffix specifies all fields, so the every field in VF06 is replaced with the addition of the corresponding fields in VF04 and VF05:

```
        VF06.X = VF04.X + VF05.X
        VF06.Y = VF04.Y + VF05.Y
        VF06.Z = VF04.Z + VF05.Z
        VF06.W = VF04.W + VF05.W
```

Because the VU0 has 4 parallel floating point units, all these additions take place in parallel.

```
         sqc2    vf6,0x0(%0)
```

Now that the result has been calculated, it must be stored. The `sqc2` instruction stands for "Save quadword from Coprocessor 2", and will save the VF06 register to the address (0x0 + `v0`).

Float to integer

```
void sceVu0FTOI4Vector(sceVu0IVECTOR v0, sceVu0FVECTOR v1) {
        asm __volatile__("
        lqc2    vf4,0x0(%1)
        vftoi4.xyzw    vf5,vf4
        sqc2    vf5,0x0(%0)
        ": : "r" (v0) , "r" (v1));
}
```

This function loads a vector, converts each element to a 12:4 unsigned fixed point integer, and saves the result.

```
        vftoi4.xyzw    vf5,vf4
```

This is the instruction that performs the conversion of each field from floating point format to 12:4 fixed point integer format. Because the VUs and the GS were designed to work together, the VU designers knew that the GS would require a 12:4 fixed point format for the X/Y coordinates. That is why the instruction is called `vftoi`**`4`**, because it creates a fixed point integer with 4 fractional bits.

There are also `vftoi0` and `vftoi12` instructions, which create fixed point integers with 0 and 12 fractional bits respectively.

Scale vector

```
void sceVu0ScaleVector(sceVu0FVECTOR v0, sceVu0FVECTOR v1, float t) {
        asm __volatile__("
        lqc2    vf4,0x0(%1)
        mfc1    $8,%2
        qmtc2    $8,vf5
        vmulx.xyzw     vf6,vf4,vf5
        sqc2    vf6,0x0(%0)
        ": : "r" (v0) , "r" (v1), "f" (t):"$8");
}
```

The `sceVu0ScaleVector` function will multiply each element of a vector (v1) by a single value (t) and store the result in vector v0.

```
        lqc2    vf4,0x0(%1)
```

The first `lqc2` instruction loads the vector to be scaled into register VF04.

```
        mfc1    $8,%2
        qmtc2    $8,vf5
```

The scale value '`t`' is represented in the C compiler by '`%2`'. When a floating point parameter is passed in a C function, it is MIPS convention to place it into the EE core's floating point unit (coprocessor 1) if possible. The `mfc1` instruction stands for "Move From Coprocessor 1", and will move '`t`' into general register 8 of the EE core. This is temporary however, as the `qmtc2` instruction ("Move To Coprocessor 2") copies it from EE core register 8 to register VF05 of VU0. The EE core registers are 128 bits wide, so the `qmtc2` instruction will copy the entire 128-bits into the VF05 register. However, the contents of the X field is the only one needed, so it does not matter that the Y, Z and W fields of VF05 are trashed.

The result of these two instructions is to move the floating point value that was on the top of the EE core's floating point stack to field X of VU0 register VF05.

```
        vmulx.xyzw      vf6,vf4,vf5
```

This instruction is called a *broadcast multiply*. A broadcast multiply will multiply all fields of the destination by a single field of the source. The source field can be any field (X, Y, Z or W). The X field is specified in the instruction above, because the instruction is called `vmulx`. The field mask is set to "`.xyzw`", so all fields in the second parameter (VF04) will be multiplied by the X field of VF05. The result is stored into VU0 register VF06. This is then saved back into memory with a `sqc2` instruction.

Multiply a matrix by a vector

```
void sceVu0ApplyMatrix(sceVu0FVECTOR v0, sceVu0FMATRIX m0,sceVu0FVECTOR v1) {
        asm __volatile__("
        lqc2    vf4,0x0(%1)
        lqc2    vf5,0x10(%1)
        lqc2    vf6,0x20(%1)
        lqc2    vf7,0x30(%1)
        lqc2    vf8,0x0(%2)
        vmulax.xyzw    ACC,   vf4,vf8
        vmadday.xyzw   ACC,   vf5,vf8
        vmaddaz.xyzw   ACC,   vf6,vf8
        vmaddw.xyzw    vf9,vf7,vf8
        sqc2    vf9,0x0(%0)
        ": : "r" (v0) , "r" (m0) ,"r" (v1));
}
```

Once a complete transformation 4x4 matrix has been generated, we multiply each vertex (in homogenous coordinates) by the matrix to get the transformed vertex. The result will be a vector that needs normalisation by W. The matrix is stored in memory like this:

Row 0	(X, Y, Z, W)
Row 1	(X, Y, Z, W)
Row 2	(X, Y, Z, W)
Row 3	(X, Y, Z, W)

A matrix is stored as an array of 4 vectors. The first vector contains the first row, the second vector contains the second row, etc.

The `lqc2` instructions load each of these vectors and the vertex into 5 registers (VF04 to VF07 for the matrix, VF08 for the vertex).

Four operations must be performed for the (matrix × vector) multiply. A full matrix multiply would look something like this (*X*d means *X*destination, *X*s means *X*source, M(a,b) means matrix row a, column b):

$$\begin{pmatrix} \text{Xd} \\ \text{Yd} \\ \text{Zd} \\ \text{Wd} \end{pmatrix} \Leftarrow \begin{pmatrix} \text{4x4} \\ \text{matrix} \end{pmatrix} \times \begin{pmatrix} \text{Xs} \\ \text{Ys} \\ \text{Zs} \\ \text{Ws} \end{pmatrix}$$

```
Xd = Xs * M(0, 0) + Ys * M(0, 1) + Zs * M(0, 2) + Ws * M(0, 3)
Yd = Xs * M(1, 0) + Ys * M(1, 1) + Zs * M(1, 2) + Ws * M(1, 3)
Zd = Xs * M(2, 0) + Ys * M(2, 1) + Zs * M(2, 2) + Ws * M(2, 3)
Wd = Xs * M(3, 0) + Ys * M(3, 1) + Zs * M(3, 2) + Ws * M(3, 3)
```

The VUs each have a 128-bit register called an *accumulator* (ACC for short). When a register is multiplied by another register, the results can optionally be directly stored to the accumulator, or added to whatever is in the accumulator. Using the broadcast multiply operation and the accumulator, we can perform this matrix multiply in 4 operations. Examine the equations above – we will be performing them all in parallel, moving from left to right.

```
vmulax.xyzw    ACC,   vf4,vf8
```

This is a broadcast multiply of field X in register VF08, which contains the vertex. VF08.X will multiply the contents of each field of VF04, and the results will be stored in the accumulator (they replace what is already there).

```
vmadday.xyzw   ACC,   vf5,vf8
```

This instruction is slightly different. It is a *broadcast multiply and add*. The broadcast field is Y, so VF08.Y multiplies each field of VF05, and the result is added to whatever is currently in the accumulator. Due to the last instruction, the accumulator contents should be VF08.X * VF04, so now the contents will be VF08.X * VF04.XYZW + VF08.Y * VF05.XYZW.

```
vmaddaz.xyzw   ACC,   vf6,vf8
```

A similar instruction also performs a broadcast multiply and add, this time multiplying VF08.Z to all elements of VF06. The result is added to the current value of the accumulator.

```
vmaddw.xyzw    vf9,vf7,vf8
```

Finally this instruction performs a broadcast multiply and add, (VF08.W * VF07.XYZW) and the result is added to the accumulator. Additionally, the final value of the accumulator is stored in register VF09.

This register is saved back to memory, and the function exits.

Summary

This program uses standard homogenous coordinates to transform a very simple 3D model into 3D space. The matrices also transform the model so that its centre is 2048.0, 2048.0 which is the middle of GS primitive space. Once transformed, the vertices are converted to standard coordinates by dividing by W. Then these vectors are placed into a GS packet on scratchpad RAM and sent to the GIF.

This program is not typical. In practice, you will probably not use the VU0 macromode libraries. There are a few reasons for this, all related to performance:

1. The overhead in moving data between the VU0 and the EE is high. The DMA can perform this much faster (although it can't move data directly into registers).
2. The VU0 instructions called from the EE can only use either the floating point units or the integer units at one time. This is less efficient than running a microprogram, which could use both units at the same time.
3. When the EE is calling the VU0 macromode instructions, there is no parallelism. The EE will wait until the VU0 is complete, and the VU0 will sit idle until it receives an instruction from the EE. You can increase the performance of your program if both the EE core and VU0 are performing tasks at the same time.

The macromode libraries are mainly there to give you a head start when experimenting with designs. They do not have the best performance and therefore would not be used in 'real' code. However, they are extremely useful for learning about the system.

In the next example, we will write a VU0 microprogram and run it.

Questions and problems

1. Q: The 4x4 matrices are arrays of vectors. If you examine the CameraToScreen matrix, you'll see a translation from an X/Y origin of (0,0) to (2048, 2048). Why is this translation necessary?

A: The GS needs the screen coordinates in "primitive space" (covered in a previous example). Recall that this space is 0 to 4095.9375 pixels for both X and Y, and part of this is mapped onto the drawing buffer. The default screen mapping is set up so that (2048.0, 2048.0) is the centre of the screen, which is where you normally want your 3D origin to be.

Section G - Example 4 – VU microcode

In this example we introduce VU microcode.

Each VU has its own code and data memory, and can run programs independently of the EE core. These programs are called *microprograms*.

A VU's microprogram can only access the data in that VU's RAM. Thus VU0 can only access VU0 RAM and VU1 can only access VU1 RAM.

The VU0 has 4KB of code RAM, and the VU1 has 16KB. Each instruction is 64 bits wide, and holds two opcodes (one for the floating point units, one for the integer units). This means that VU0 and VU1 can hold a maximum of 512 and 2048 instruction lines respectively.

The ability of the VUs to run their microprograms independently of the EE core and of each other means that the PlayStation 2 can use the EE core, the VU0 and VU1 in parallel, which increases performance.

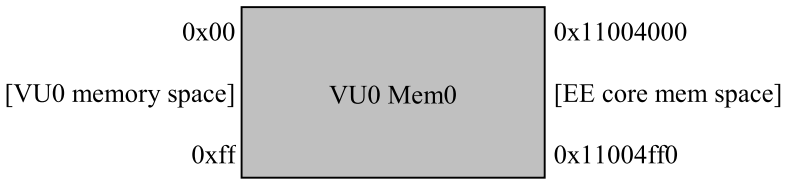
[Note: In general, the DMA and VIFs are used to copy microprograms to VU memory. This is the 'correct' method, and is used in the next example. In this example however, the microprogram is simply copied to VU0 RAM. This is a less efficient method, but is simpler to explain.]

This example uses a VU0 microprogram to perform all the 3D calculations. The EE core program works by copying the microprogram to VU0 code RAM, copying the data to VU0 data RAM, and activating the microprogram. Once the microprogram is complete, the results are used to build a DMA list to send through the GIF to the GS, as performed in the previous examples.

Accessing VU0 memory

When the VU0 is not running microprograms, its memory is accessible by the EE core. The VU0 has 4KB of code RAM and 4KB of data RAM. The VU0 code RAM is called MicroMem0 and is mapped to addresses 0x11000000 to0x11000ff0. VU0 data RAM is simply called Mem0 and is mapped from 0x11004000 to 0x11004ff0. See section 4.2 in the EE Users Manual for a complete description of the VU RAM address mappings).

Because the VU0 deals exclusively with 128 bit registers, it can only read or write to 128 bit aligned addresses (addresses where the lower 4 bits are 0). In VU0 memory address space, address 0 maps to RAM address 0x11000000, and address 1 maps to RAM address 0x11000010. Address 1 does not map to 0x11000001 as you might expect. Therefore incrementing a VU0 pointer by 1 actually moves the pointer 128 bits, not 8 bits.



We will use a function called `copyAlignedQWords()` to move data in and out of VU0 memory. This is highly inefficient, but it is useful for this example because it means there is less new material to cover. The next example shows the most efficient way to copy memory to a VU, where the DMA to VIF is introduced.

Before we begin the walkthrough, some design decisions need to be made. The VU0 program needs to be placed somewhere in VU0 MicroMem0, and it must get its data from somewhere in Mem0. We will hardcode these addresses.

The VU0 microprogram will be placed at MicroMem0 address 0 (this maps to 0x11000000) in EE core space).

The transformation matrix will be placed at Mem0 address 0 (this maps to 0x11004000). The 4 vertices will be placed directly after this at address 4 (0x11004040). The transformed vertices will be written by the microprogram to address 8 and upwards (0x11004080).

Walkthrough – VU program

This file is assembled into an object file using the VU assembler, called `ee-dvp-as`. You can find documentation on this assembler in `/usr/local/sce/ee/gcc/doc/ee.pdf`.

It will create a file that we can link into the main EE core program. The linker will place it at an address somewhere in EE RAM. It will **not** be automatically placed in VU0 memory. Instead, it will be stored with the rest of the program in EE memory. The program has to manually upload the program and the data to the VU0 before it can be used.

```
        ; these are labels we will want to access externally
    .global Vertices
    .global myVU0microprogram
    .global endOfProgram

    .p2align 4  ; This aligns the following data to quadword alignment
```

The last instruction sets the alignment of the following data to quadword alignment. We need to align the data to quadword alignment for two reasons.

Firstly, it is not possible for the EE core to read/write arbitrary sizes of VU0 memory. It is only possible to read/write quadwords, and these quadwords must be quadword aligned. When we copy the data into VU0 memory, we will use a simple function that loads a quadword, then stores a quadword, then increments the source/destination pointers. In order to load the quadword, it must be quadword aligned, and that is why the `.p2align 4` directive is used.

Secondly, the next example uses the DMA to transfer the data into the VU0 instead of having to copy it manually. Because all DMA data must be quadword aligned, we must quadword align our VU microprogram and data.

```
Vertices:
    .float  -100, -100, 0, 1
    .float   100, -100, 0, 1
    .float  -100,  100, 0, 1
    .float   100,  100, 0, 1
```

The 4 vertices of the quadrilateral are defined here. The `.float` directive interprets the values that follow to be single precision floating point numbers, and stores them contiguously in memory. The `.float` directive does not know about alignment, so it is good practice to use the alignment directives to ensure the data is quadword aligned.

This data will eventually be copied to VU memory by the main EE core program before the main loop. The microprogram will read the vertices from VU0 memory and transform them, then store the results in another area of VU0 memory.

```
    .vu
```

This directive sets the assembler into a mode where it can assemble VU instructions. You must use this directive directly before using VU opcodes.

```
myVU0microprogram:
    NOP                                 IADDIU      VI01, VI00, 4
    NOP                                 IADDIU      VI02, VI00, 0
    NOP                                 IADDIU      VI03, VI00, 4
    NOP                                 IADDIU      VI04, VI00, 8
```

This is the beginning of the program. Recall that the object file created by the assembler will be linked in as part of the program, and will be placed somewhere in EE RAM, not VU0 RAM. Therefore the labels in this object file will point to addresses in EE RAM.

There are two instructions per line. The instructions on the left are the floating point unit instructions, and the instructions on the right are the integer unit instructions. For simplicity, this example does not execute a floating point and integer instruction at the same time in this program, although it could be rewritten so that it did.

The integer unit has sixteen 16-bit registers, named VI00 to VI15. VI00 always contains the value 0, even if you write to it.

There is no "Load Immediate" instruction, so to get an arbitrary number into a register, you must use the IADDIU instruction, which does have an immediate argument. This function will add the immediate argument to a register, and store the result in another register. Since VI00 is always 0, we can load an arbitrary register with an arbitrary value with the instruction:

```
IADDIU VI<destination register>, VI00, <value>
```

Here we load the registers VI01 to VI04 with the values 4, 0, 4 and 8 respectively.

We will use register VI01 as a loop counter. We will be processing 4 vertices, so load 4 into this register.

VI02, VI03 and VI04 are used as Mem0 pointers to the transformation matrix, the input vertices and the output vertices respectively. Recall that the VU0 can only read or write quadwords, so pointer values are in units of quadwords. VU0 refers to its memory as addresses 0 through to 0xff. These map to EE core addresses 0x11004000 to 0x11004ff0.

VI02 points to VU0 address 0 (0x11004000), VI03 points to VU0 address 4 (0x11004040), and VI04 points to VU0 address 8 (0x11004080). Incrementing a VU0 pointer by 1 moves it forward by a single quadword.

You may be surprised that macros have not been used for better legibility. Surely it is easier to remember "`MATRIX_POINTER`" than "VI02"? However, the `ee-dvp-as` assembler is very rudimentary and does not support such macros. If you want to use macros for better legibility, it is possible to set up your Makefile to run the file through a pre-processor (such as the GCC C pre-processor), which will perform macro expansion before passing it to `ee-dvp-as`.

[A word of warning: `ee-dvp-as` is very strict about syntax. Extra whitespace can cause problems]

```
NOP                                 LQI.xyzw    VF04, (VI02++)
NOP                                 LQI.xyzw    VF05, (VI02++)
NOP                                 LQI.xyzw    VF06, (VI02++)
NOP                                 LQI.xyzw    VF07, (VI02++)
```

These instructions load the transformation matrix into the registers.

The `LQI` instruction means "Load Quadword Increment". This means that the VU0 memory address pointed to by the source register (in this case, VI02) is read, and the contents placed into the destination. Then the source register is incremented, so it will point to the next quadword.

The `LQI` instruction also has a field mask (`.xyzw`). These specify which fields should be written in the destination register. If a field is not present in the mask, that field in the destination register is left untouched.

The opcode itself appears to have some redundancy – that is, there are two pieces of information that inform you that this instruction should increment the source register. The first is the instruction name itself (`LQI`, where the 'I' means Increment) and the second is the '`++`' in the source register. Surely you should only need one of these? Unfortunately, it is one of the idiosyncrasies of the VU assembler that some instruction definitions contain redundant information like this. We will see a few more examples of this later on in the program.

Hardcoding addresses

Probably the most unusual aspect of these instructions is that they use hardcoded addresses – addresses specified by raw numbers defined by the programmer, as opposed to addresses automatically calculated by the linker. For example, the VU0 address of the matrix (0) is specified with the instruction "`IADDIU VI02, VI00, 0`". This effective hardcodes the address of the matrix to VU0 address 0. Indeed, the start address of the vertices is hardcoded to be 4, and the start address of the transformed vertices is hardcoded to be 8.

Hardcoded addresses are used because there is no pre-defined data in the VUs when the program begins. It is up to the program to place information there. The program has complete control over the code, data and register contents of the VU.

```
loop:
    NOP                                 LQI.xyzw    VF01, (VI03++)
```

This is the beginning of the main loop. The first instruction loads in the next vertex, pointed to by VI03. VI03 is incremented afterwards. The vertex is put into floating point register VF01.

```
    MULAx       ACC, VF04, VF01x        NOP
    MADDAy      ACC, VF05, VF01y        NOP
    MADDAz      ACC, VF06, VF01z        NOP
    MADDw       VF02, VF07, VF01w       NOP
```

These four instructions are the matrix/vector multiply that was covered in the previous example. The result is put into register VF02.

```
    NOP                                 DIV Q, VF00w, VF02w
    NOP                                 WAITQ
```

The vector needs to be renormalised, by dividing by W. Instead of dividing each element by W, it is more efficient to scale the entire vector by 1/W. The `DIV` instruction calculates Q = 1/W'. Q is a special VU0 register that holds the result of the floating point divide instruction.

VF00 always contains the values (0, 0, 0, 1). To calculate 1/W', the register field VF00.W is specified as the value to be divided. The divisor is VF02.W. The `DIV` instruction begins this operation, which takes more than one cycle. Because the divide operation may take some time, we use the `WAITQ` instruction to stall the program until it is complete. When it moves on, the Q register will contain 1.0/W'.

```
    MULQ.xyzw   VF02, VF02, Q           NOP
```

Now we multiply each field in the transformed vector VF02 by Q. The result is stored in the same register.

Once again there some redundancy in the instruction. The opcode `MULQ` specifies that the value will be multiplied by Q, but it is also necessary to specify Q as one of the parameters. This doesn't mean that you can pick any other register for a `MULQ` instruction – we must use Q. This redundancy is a quirk of the assembler.

```
    FTOI4.xyzw  VF02, VF02              NOP
```

This performs the conversion from floating point to 12:4 fixed point integer format. The transformation matrix was chosen so that the vertices would lie well inside the range (0, 4095.9375), which is GS primitive space.

```
    NOP                                 SQI.xyzw VF02, (VI04++)
```

The `SQI` instruction is very similar to the `LQI` instruction. It will save the result to the address pointed to by VI04, then increment VI04.

```
    NOP                                 ISUBIU VI01, VI01, 1
    NOP                                 NOP
    NOP                                 IBNE VI01, VI00, loop
    NOP                                 NOP
```

This is the end of the loop. The first instruction will decrement VI01, which holds how many vertices left to process. Then the IBNE tests VI01 to see if it does not equal VI00 (which is always 0). If so (i.e. there are vertices left to process) then it will branch back to label `loop`.

You may have noticed that a label was used to branch to. If the labels all refer to areas in EE RAM, then won't the branch address be incorrect? The reason why this works is that all VU branches are relative to the position of the program counter ('PC relative'). Rather than branching to an explicit address, the program will branch to an address relative to where the branch instruction is. This means that no matter where the code is uploaded to in VU0 MicroMem0, branches will always work.

There are also two lines in there that have two `NOP` instructions in them. They are there for a reason.

The first pair of `NOP` instructions is to act as padding. If an operation is performed on a register, branch instructions must wait at least one cycle before performing tests on that register. This is because of the way the pipelining works inside the VUs.

The second pair are there to pad the *delay slot*. Because of the way the pipelining works, the instruction line after a branch is always executed, even if the branch is taken. This instruction cycle is called the delay slot and is also found on the pipelines in MIPS processors like the EE core and IOP. Sometimes it is possible to fit useful instructions into the delay slot, but in this program it is not, so `NOP` instructions are used instead.

```
    NOP[e]                              NOP
endOfProgram:
```

When the program finally processes all the vertices, it will 'fall through' the conditional branch and continue on. It will execute this instruction pair, which performs nothing. However, notice the `[e]` next to the `NOP` on the left. This will halt the VU0 microprogram.

There is no 'halt' instruction on a VU. Instead, each instruction line has an 'end' bit field. If this field is set to 1, then the VU will execute that instruction and halt. The 'end' field can be set on any instruction line, not just `NOP` lines. Writing `[e]` on the right of the leftmost instruction will set the 'end' field.

We use the `endOfProgram` label to calculate the length of the program. This length is used when the EE core uploads it to MicroMem0.

Walkthrough – C program

```
extern u_long Vertices __attribute__((section(".vudata")));
extern u_long128 myVU0microprogram __attribute__((section(".vudata")));
extern u_long128 endOfProgram __attribute__((section(".vudata")));
```

When the EE program begins, the VU microprogram and its data are not yet in VU0 memory. They need to be uploaded manually. These external references allow this object file to access the VU program in EE memory so it can be uploaded.

The VU0 destination addresses are hardcoded. To reiterate:

• Microprogram	MicroMem0 address 0	(0x11000000)
• Transformation matrix	Mem0 address 0	(0x11004000)
• Vertices	Mem0 address 4	(0x11004040)
• Transformed vertices	Mem0 address 8	(0x11004080)

```
sceVu0FMATRIX                         // Vertex transformation matrices
    LocalToWorldMatrix,
    WorldToScreenMatrix,
    CameraToScreenMatrix,
    WorldToCameraMatrix;
```

The various transformation matrices. Notice that LocalToScreenMatrix is not defined, which is the final transformation matrix. This is because the program calculates it every frame and saves it directly to VU0 memory.

```
void copyAlignedQWords(u_long128 *dest, u_long128 *src, int numQWords);
```

This function will copy a number of quadword aligned quadwords from `src` to `dest`. It is used to copy data and programs to and from VU0 memory. Please note that this is not the usual way of performing this operation. Normally the program uses the DMA to transfer data to the VUs, but some extra concepts have to be introduced before that can be shown.

```
        // Copy VU0 microcode into MicroMem0
    copyAlignedQWords((u_long128 *)0x11000000, (u_long128 *)&myVU0microprogram,
        (((u_int)&endOfProgram - (u_int)&myVU0microprogram) + 15) >> 4);
```

The microcode is copied into VU0 MicroMem0. The first argument is the destination in EE core space (see above), and the second is the source address (the address of the myVU0microprogram label in EE main memory).

The 3rd argument is the number of quadwords to transfer. Because an instruction is 64 bits, the program's total length may not be exactly a multiple of a quadword. The correct number of quadwords to transfer is the difference between the end and the start addresses plus 15, divided by 16.

Now the program has been uploaded to memory. A program can only be copied to a VU in this way when the VU is not running any programs. The program has complete control over the VUs - the code is only uploaded once, since it will never be overwritten.

```
        // Copy vertices into Mem0
    copyAlignedQWords((u_long128 *)0x11004040, (u_long128 *)&Vertices, 4);
```

The program needs the 4 vertices to transform. For convenience, these were defined along with the program. They are copied to Mem0, to the address decided on previously.

```
            // Create the final Local->Screen matrix
        sceVu0MulMatrix((void*)0x11004000, WorldToScreenMatrix, LocalToWorldMatrix);
```

The LocalToScreen matrix is calculated here. The destination parameter (argument 1) is set to the Mem0 address where the microprogram will read the transformation matrix when it is started.

Now the microprogram is uploaded into MicroMem0, and the transformation matrices and the 4 vertices are in the correct place in Mem0. The microprogram is ready to begin.

```
        asm("vcallms 0");   // Micromem addr divided by 8
```

This instruction activates the VU0 microprogram. The parameter is the VU0 MicroMem0 address to begin execution. The microprogram was copied to MicroMem0 starting at address 0, and that is where it should begin execution. Because the VU instructions are 64 bits wide and not 128 bit, the start address is in units of 64 bits, not 128 bits.

Now the microprogram is running. Because the microprogram runs in parallel to the EE core, the EE core is free to perform other tasks now if it wishes. As it happens, there are no other tasks to perform, so the EE core simply waits for the VU0 microprogram to finish.

```
        asm("vnop");
```

This instruction seemingly does nothing, but it does serve a purpose. If the VU0 is running a microprogram and the EE core attempts to execute a macromode instruction, the EE core will stall until the VU0 microprogram is complete. In this case, the EE core wants to wait until the microprogram has completed execution, so it attempts to execute a `vnop` instruction, which stalls the program until the VU0 microprogram is complete.

```
            // Copy transformed vertices
        copyAlignedQWords((u_long128 *)spr, (u_long128 *)0x11004080, 4);
```

Now that the program is complete, the 4 transformed vertices should be in Mem0, starting at address 8 (0x11004080). The transformed vertices are ready to be put into the DMA packet. This call copies the 4 vertices onto the end of the DMA packet, which is then sent to the GIF.

Summary

This example copies a VU microprogram and vertex data to VU0 code and data RAM respectively. The main loop calculates a 3D transformation matrix and copies it to VU0 data RAM, then activates the VU0 microprogram which transforms the vertices already in VU0 RAM. The EE core waits for the program to complete, then copies out the results. These are placed into the GS packet which is transferred by DMA to the GIF.

In the next example, the DMA is used to upload code and data into the VU0, and is also used to activate the program.

Questions and problems

1. Q: This is all fine for such a simple 4 vertex model, but what about something with 2000 vertices that won't fit into VU memory all at once?

A: If a program has a large amount of data to process, it will have to split that data up into smaller manageable chunks. It's unlikely that that the program would ever need all 2000 vertices at once inside the VU0.

The program would upload smaller chunks to the VU, which would process them. When complete, the results are read out, and the next chunk is uploaded. This can be managed without EE core intervention, which we'll cover in the next example.

2. Q: The VU microprogram doesn't set the W field of the transformed vertex to 0 before storing it. Wouldn't this possibly set a random value to the ADC bit?

A: When the field is normalised by multiplying by 1/W, the W field contains W, and therefore the resulting value of the W field will be 1.0 (or close to it, allowing for an error caused by lack of precision). When this is converted to integer format, 1.0 translates to 0x10. This has no chance of touching the ADC bit, which is bit 15 in the W field. Therefore the ADC bit will be 0 unless explicitly set by the program.

3. Q: How fast is it to upload code?

A: That depends on the size of the code to upload of course, but in general it is just as fast as uploading data. In a 'real' application, the VU code could be rewritten thousands of times per frame.

Section H - Example 5 – The VIFs

In this example we introduce the VIFs, and use the DMA to upload and run the VU0 microprogram.

Each VU has an Vector unit InterFace (VIF). The VIF attached to VU0 is called VIF0 and the VIF attached to VU1 is called VIF1.

There are two DMA channels dedicated to moving data from EE core memory to the two VIFs. These channels are referred to by the libraries as `SCE_DMA_VIF0` and `SCE_DMA_VIF1`. As is the case with the GIF, the DMA sends data to a VIF in units of quadword.

The VIFs interpret these quadwords as commands and command data. These commands can be used to transfer programs and data to VU memory. They can also activate microprograms, and stall the DMA until the microprograms are complete.

The VIFs are quite sophisticated. They can perform complex operations on the incoming data, so that the data is transformed into the correct format for the VUs. Some data can be decompressed by the VIFs. This reduces the amount of data that has to be transferred by the DMA, which will increase performance.

The ability of the DMA and the VIFs to upload programs and data to a VU, then activate microprograms and wait for completion, allows a VU to work almost autonomously.

For example, a DMA/VIF list could be created to work in the following way:

1. Upload program to VU0.
2. Upload data set X to VU0.
3. Activate VU0 microprogram [Program works on data set X]
4. Wait for microprogram to complete.
5. Request interrupt [This would start an EE core subroutine that copied out results of processing data set X. The DMA and VIF continues while this is happening]
6. Upload data set X+1.
7. Activate program [Program works on data set X+1].
8. Etc…

If a DMA/VIF list could be designed in such a way, then the VU0 would be ‘fed’ data, and the EE core would copy out the results when each data set was processed. This means the EE core could be performing other useful tasks while the VU0 processing is taking place. The EE would only be interrupted to copy out intermediate results.

If this example was applied to the VU1, which has the ability to send data directly to the GIF via PATH1, then the EE core does not need to intervene at all (except to kick off the initial DMA transfer). A single DMA transfer could potentially send all the necessary programs and data to the VU1 without any EE core intervention. The VIF takes care of stalling the DMA at appropriate times so the transfer does not happen too quickly. While this is happening, the VU0 and EE core could be performing other tasks. This method is used in example 6.

In this example, the VIFs are introduced, and the program is uploaded and activated by the DMA.

In Example 4, we copied the microprogram code and data to the VU0 using the EE core. In general, this is not how code and data are usually moved between the VUs and EE main memory, because it is not efficient. Instead, we use the DMAC and the VIF0 (Vector unit InterFace) to move the data. This frees the EE core for more useful work.

VIF codes

Each VU has its own VIF. This example uses VIF0, which is attached to VU0. Special codes called VIF codes are embedded into the data to be transferred to a VIF. The VIF codes are similar in function to the GIF Tags – they tell the VIF about the data that follows them, and where to place it in VU RAM.

They can also perform operations like activating a VU program, or stall the DMA until a VU program is complete.

A VIF code is always 32-bits wide. Some VIF codes have data that follows them, and the data is always in units of 32-bits, even though they are transferred a quadword at a time by the DMAC.

One of the VIF's most important VIF codes is the VIF UNPACK mode command. When data arrives at the VIF destined for VU data memory, the VIF can unpack the incoming data into quadwords. For example, a VIF UNPACK command can expand a single 32 bit word into an entire quadword. If your data can be packed, then this will help reduce the amount of data that is transferred from EE memory. The less data transferred by the DMA, the more bandwidth the EE core has to main memory. This will speed up DMA transfers and the EE core programs. In this example, the simplest form of the UNPACK code is used, where the incoming data is not changed by the VIF. The data that arrives is directly written to the destination quadword.

How the program works

This program works in a very similar way to Example 4. However, the DMA transfers the microprogram to MicroMem0 instead of the EE core. The vertices and microprogram are uploaded to the VU0 and will be in VU0 RAM when the main loop begins.

During the main loop, the program calculates the new transformation matrix. This is then embedded into a DMA list that is sent to the VIF0, which places it in VU0 RAM. A VIF code is embedded in the DMA list after the matrix that will activate the VU0 microprogram.

The EE core program sends this DMA list, then waits for both the DMA transfer and the microprogram to end. When the microprogram ends, the vertices will have been transformed and will be in VU0 RAM.

The results are copied out of VU0 RAM onto a DMA list in scratchpad RAM. This is then transferred to the GIF by the DMA, in the same way that was used in the previous examples.

Walkthrough – VU assembler source

The `ee-dvp-as` assembler can also assemble DMA tags, VIF codes and GIF tags. This example uses the assembler to create the DMA tags and VIF codes, because they will be static in this example, and therefore do not need to be dynamically created.

```
    .global MatrixAndStart
    .global LocalToScreenMatrix
    .global ProgramAndData
```

These `MatrixAndStart` label references the first DMA tag. This tag transfers the transformation matrix and the VIF codes that activate the microprogram.

The `ProgramAndData` label references the DMA tag that transfers the microprogram and the vertices.

TTE bit

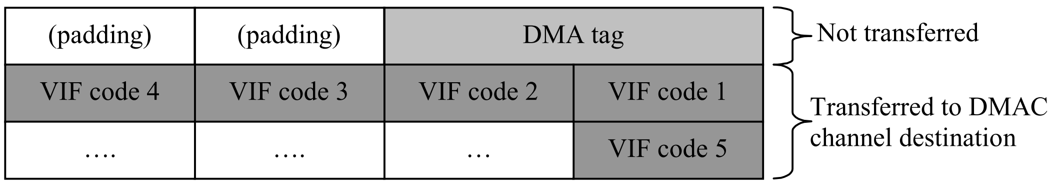
```
    .DmaPackVif 1    ; specifies that the DMA tag will be packed with VIF codes
```

The DMA tag is 8 bytes wide, but must be aligned on a quadword (16 byte) boundary. This leaves the upper 8 bytes of the quadword containing the DMA tag unused. To use this space, each VIF channel has a TTE (Transfer Tag Enable) bit in the appropriate DMA channel's control register. When TTE is set to 1, the DMAC will transfer the quadword containing the DMA tag to the peripheral, before transferring the tag's data. The peripheral is intelligent enough to ignore the tag itself, but will use the top half of the quadword. This allows the user to embed VIF codes in the upper 2 words of the DMA tag quadword. This saves space and is more convenient when using a REF tag (we'll see how later).

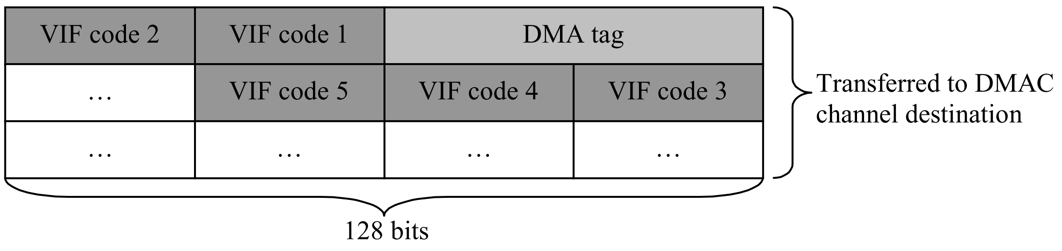
The assembler cannot not know the state of the TTE bit in advance (because the TTE bit is in a control register) so the `.DmaPackVif` directive specifies the state of the TTE bit in the source. For this example, the TTE bit will be set to 1 by the EE core program, so the VU assembler source sets `.DmaPackVif` to 1 also.

If `.DmaPackVif` was set to 0, the assembler would pad the DMA tag quadword before assembling any VIF codes. As it is set to 1, the VIF codes to follow are be packed into the top half of the DMA quadword.

TTE = 0 (.DmaVifPack 0)



TTE = 1 (.DmaVifPack 1)



```
MatrixAndStart:
    DMAend *
```

This is the DMA tag that will transfer the transformation matrix to the VU0 and start the VU0 microprogram. The assembler understands all 8 different types of tags. In this case, we are using a DMA tag with an ID of 'END', which is why the instruction is `DMAend`. The parameter is an expression specifying the length of the transfer. It is possible to specify either a value, or a '`*`'. In the latter case, the assembler will automatically calculate the length of the transfer using the next `.EndDmaData` directive.

```
    UNPACK 4, 4, V4_32, 0, *
```

This is a VIF UNPACK assembler opcode. The UNPACK code tells the VIF about data that follows the UNPACK code, and where it should be placed in VU memory. It specifies:

- What input format the data is in.
- Where the data should be placed in VU memory.
- How many quadwords it will unpack to (which, together with the input format, implies the amount of data to follow).

The last three arguments of the UNPACK assembler directive specify these three parameters.

The UNPACK VIF code can be set to any single input format, but it always unpacks in units of quadwords. The input format we use (`V4_32`) is the simplest UNPACK instruction. This tells the VIF to expect four 32 bit words as input (hence the name `V4_32`), and to directly copy these without change to the output quadword.

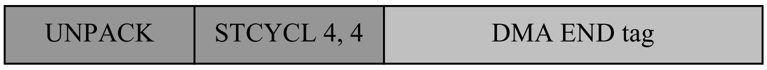
The address we want to place this data in Mem0 is VU address 0, so the 4th parameter is set to 0.

We let the assembler automatically calculate the length of the data to be unpacked by using a '`*`' as the 5th parameter. The assembler calculates this length based upon the next `.EndUnpack` directive.

The first two parameters are the WL/CL register values. This is because the assembler cannot interpret the UNPACK data without knowing the state of the VIF's WL and CL registers (these are covered in detail in the EE Users Manual).

In fact, the assembler will create two VIF codes from this assembler opcode. The first VIF code is STCYCL, which sets WL and CL to the values specified by the first two parameters. The second VIF code is the actual UNPACK code.

So far, the assembler has created the following structure:



UNPACK	STCYCL 4, 4	DMA END tag

```
LocalToScreenMatrix:                ; // Transform matrix
    .float  0, 0, 0, 0
    .float  0, 0, 0, 0
    .float  0, 0, 0, 0
    .float  0, 0, 0, 0
```

This is the area used to store the LocalToScreenMatrix. We will calculate and update this space during the program, so these values are just placeholders. After calculating the matrix and storing it here, we'll use the DMA to upload to the VU0.

```
    .EndUnpack
```

This directive helps the assembler automatically calculate the number of quadwords to UNPACK in a previous UNPACK instruction. If the "length" parameter of UNPACK is '`*`', then the assembler will search for a `.EndUnpack` directive to calculate the length.

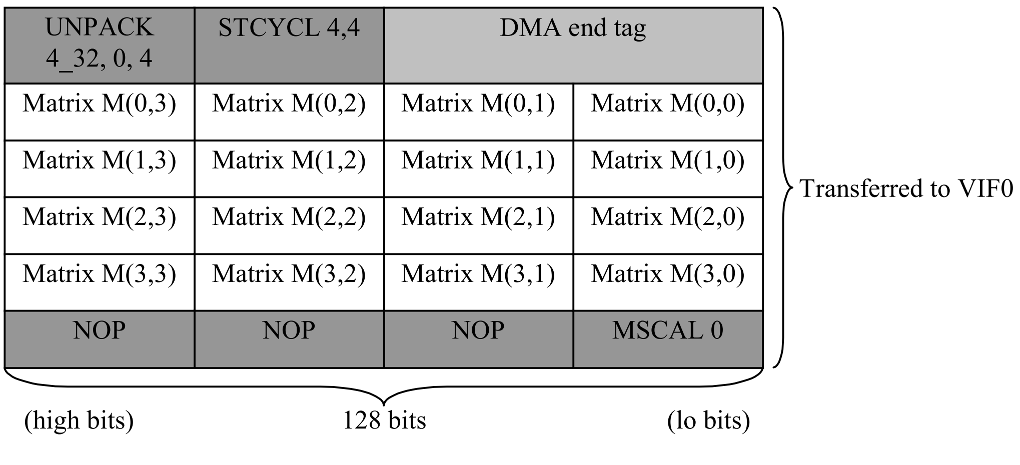
```
    MSCAL 0             ; // kick off program
```

This VIF code will activate a microprogram. The parameter is the VU address to begin execution. Once again this is a hardcoded address.

```
    .EndDmaData
```

Similar to the `.EndUnpack` directive, the `.EndDmaData` directive helps the assembler automatically calculate the length of the DMA transfer. If the transfer does not end on a 16 byte (quadword) boundary, the assembler will pad the last quadword with VIF NOP codes.

The DMA list that the assembler has just created now looks like this:



UNPACK 4_32, 0, 4	STCYCL 4,4	DMA end tag	
Matrix M(0,3)	Matrix M(0,2)	Matrix M(0,1)	Matrix M(0,0)
Matrix M(1,3)	Matrix M(1,2)	Matrix M(1,1)	Matrix M(1,0)
Matrix M(2,3)	Matrix M(2,2)	Matrix M(2,1)	Matrix M(2,0)
Matrix M(3,3)	Matrix M(3,2)	Matrix M(3,1)	Matrix M(3,0)
NOP	NOP	NOP	MSCAL 0

Although the DMA tag itself is transferred with the rest of the data (because the DMA channel's TTE bit will be set to 1), the VIF is intelligent enough to ignore it.

As the diagram shows, the assembler has created two VIF codes from the UNPACK assembler opcode (STCYCL and UNPACK). These have been placed in the upper 64 bits of the DMA tag quadword because '`.DmaVifPacket 1`' was specified.

```
ProgramAndData:
```

```
DMAend *
UNPACK 4, 4, V4_32, 4, *     ;// Vertices
.float  -100, -100, 0, 1
.float   100, -100, 0, 1
.float  -100,  100, 0, 1
.float   100,  100, 0, 1
.EndUnpack
```

This is a new DMA list that uploads the vertices and the microprogram to VU0. It is much the same as the first DMA list. The `UNPACK/.EndUnpack` pair place the vertices at VU0 Mem0 address 4, using the same method used in the first DMA list.

```
MPG 0, *         ; Upload program to address 0
```

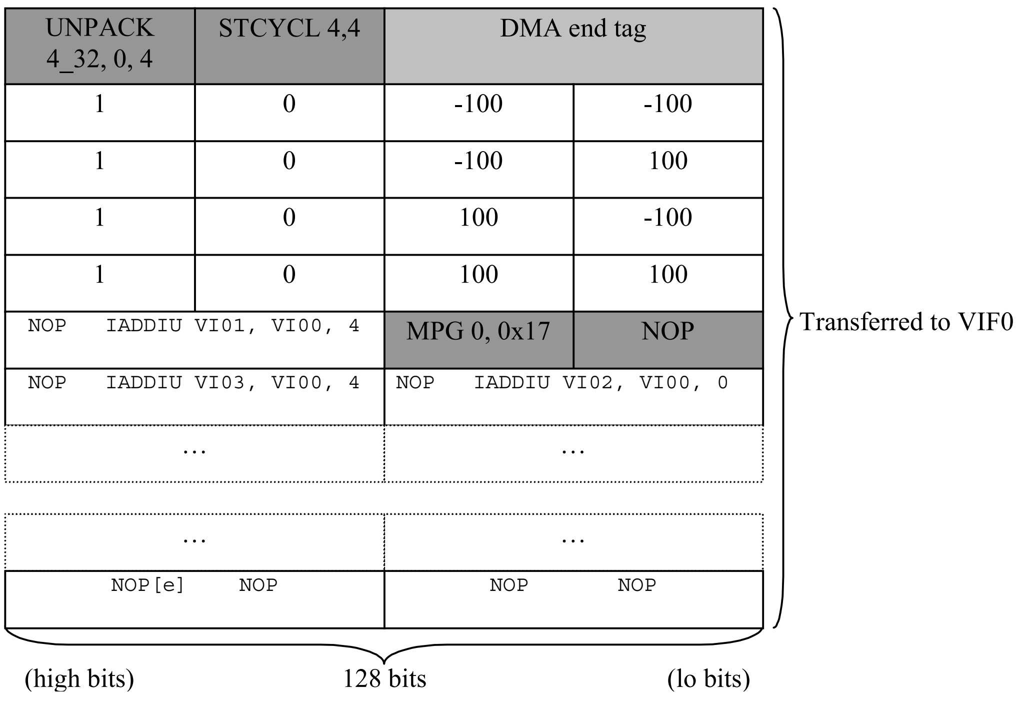
This VIF code begins transmission of a VU0 microprogram. It takes two parameters, the MicroMem0 start address of the program, and the length of the program to follow. Once again we use the '*' and a `.EndMpg` directive to automatically calculate the length.

`MPG` is the only VIF code where the memory alignment of the data to follow is important. Normally, data that follows a VIF code only has to be word aligned (4 bytes). However, `MPG` data must be 8 byte aligned. The assembler knows this, and will insert a `NOP` just before the `MPG` instruction so that the program data is aligned to 8 bytes.

```
NOP                                 IADDIU      VI01, VI00, 4
NOP                                 IADDIU      VI02, VI00, 0
NOP                                 IADDIU      VI03, VI00, 4

 …                                    …

NOP                                 IBNE VI01, VI00, loop
NOP                                 NOP
NOP[e]                              NOP

.EndMPG
.EndDmaData
```

The program is unchanged from Example 4.

This creates the following DMA list:



UNPACK 4_32, 0, 4	STCYCL 4,4	DMA end tag	
1	0	-100	-100
1	0	-100	100
1	0	100	-100
1	0	100	100
NOP IADDIU VI01, VI00, 4		MPG 0, 0x17	NOP
NOP IADDIU VI03, VI00, 4		NOP IADDIU VI02, VI00, 0	
…		…	
…		…	
NOP[e] NOP		NOP NOP	

Walkthrough – C program

```
extern sceVu0FMATRIX LocalToScreenMatrix __attribute__((section(".vudata")));
extern u_long128 MatrixAndStart __attribute__((section(".vudata")));
extern u_long128 ProgramAndData __attribute__((section(".vudata")));
```

These external references refer to the transformation matrix within the DMA list defined in the VU listing above, and the addresses of the predefined DMA lists.

```
    dmaGif = sceDmaGetChan(SCE_DMA_GIF);
    dmaVif0 = sceDmaGetChan(SCE_DMA_VIF0);
    dmaVif0->chcr.TTE = 1;
```

Two DMA channels are used in this program. The first is the GIF channel, the one used in all previous examples. The second is the VIF0 channel, used to transfer the data and program to VU0.

The last line sets the TTE bit in the control register of the VIF0 channel. This will transfer the tag along with the rest of the data for the VIF0 channel only.

```
        // Transfer vertices and VU0 microcode into MicroMem0
    sceDmaSend(dmaVif0, &ProgramAndData);
```

The program and vertex data only have to be uploaded once, so this is performed here before the main loop. The VIF codes take care of placing the data in the correct area of VU RAM.

Uncached memory access

```
    pLocalToScreenMatrix =
        (sceVu0FMATRIX *)(((u_int)&LocalToScreenMatrix & 0x0fffffff) | 0x20000000);
```

The transformation matrix will be created and uploaded every frame. The EE core address where the matrix is loaded from is `&LocalToScreenMatrix`. This function initialises a pointer to that address. The binary operations make sure that the high nibble of the address is set to 0x2. This places the pointer in uncached memory.

The EE has 3 ways to access the same address in main memory. These are *cached*, *uncached*, and *uncached accelerated*. The type of access is determined from the value of the address' high nibble. If the high nibble is 0x0, then the address is considered cached. If it is 0x2, then it is uncached. If it is 0x3, then it is uncached accelerated.

For example, address 0x00010000, 0x20010000 and 0x30010000 all map to the same address in EE main memory, however the first is cached, the second is uncached and the third is uncached accelerated. [Scratchpad RAM lies at 0x70000000 and is considered to be uncached].

When cached memory is read, it is loaded into the cache if it is not already there. When it is written, it is not stored back to main memory until it is flushed. A flush happens according to the caching algorithm inside the EE core, or explicitly due to a `FlushCache(0)` call. If cached memory is written to and `FlushCache` is not called, it is not guaranteed that main memory is consistent with the current state of the cache.

When uncached memory is written to, the data is immediately stored back to memory and to the cache. It is used when creating or adjusting data that will be transferred by DMA, since the DMA does not know about the cache. This is why it is used in this example.

'Uncached accelerated' memory works in a different way. There is a separate cache, only one cache line wide. When the data is read from an uncached accelerated address, it is also read into this cache line if it is not already present. Subsequent reads from the same address or addresses within the same cache line will access the one line cache instead of main memory. This allows faster processing of sequential data without destroying the existing contents of the main cache.

Because the program adjusts a DMA list, and the DMAC knows nothing of the cache, the program uses an uncached address to refer to the transformation matrix. When data is written to this address, the data will be stored immediately. This means a `FlushCache(0)` call is unnecessary, which is better for performance as `FlushCache` is very slow and should be avoided where possible.

```
            // Create the final Local->Screen matrix
        sceVu0MulMatrix(*pLocalToScreenMatrix,
            WorldToScreenMatrix, LocalToWorldMatrix);
```

Here the matrix is calculated and saved into the DMA list.

```
            // Upload matrix and activate program

        asm("sync.l");
```

Once again the `sync.l` instruction flushes the EE core's write buffer, which is necessary before setting off a DMA (to ensure memory is consistent with the EE's perception of it). This is necessary even when uncached memory is used.

```
        sceDmaSend(dmaVif0, &MatrixAndStart);
```

The DMA to the VIF0 is started. This will upload the matrix to Mem0 and activate the program.

```
        spr = (u_long *)SPR;
        *(spr++) = squareGIFTag[0];
        *(spr++) = squareGIFTag[1];
        *(spr++) = polyColour[0];
        *(spr++) = polyColour[1];
```

Here we see the first example of parallelism. While the DMA transfer to the VIF0 is taking place, the EE core starts building the DMA list. Both operations happen at the same time. By running tasks like this in parallel, we can get much greater performance out of the EE.

```
            // Wait for DMA and program to complete
        while(dmaVif0->chcr.STR == 1);
        asm("vnop");
```

Before the final DMA transfer can take place, the EE core needs to copy the results out of VU0 memory. To make sure the VU0 has completed its task, a loop waits for the DMA to finish (the `STR` bit in the channel's control register will be 0 when the DMA channel is not running). Then a "`vnop`" macromode instruction is used, which stalls the EE core until the VU0 has completed executing any microprograms. There are more efficient ways to check whether the VU0 has completed its task, but they are not covered here.

It's necessary to perform both checks. Here's why:

If only the first check was made (DMA complete), then it's possible that the DMA was complete but the VU0 microprogram was not. In this case the EE core would have attempted to access data while the microprogram was still running, which is not permitted.

If only the second check was made (microprogram complete), then it's possible that the "`vnop`" could have been called while the data was still uploading, which would have returned immediately because the microprogram had not yet begun.

It is good practice to keep synchronisation issues like this in mind when trying to run two or more processors in parallel.

```
        copyAlignedQWords((u_long128 *)spr, (u_long128 *)0x11004080, 4);

        asm("sync.l");
        sceDmaSendN(dmaGif, (u_long128*)(SPR | 0x80000000), 6);
    }
```

Now that we are confident that the DMA transfer has completed and the VIF0 program has ended, the transformed data is copied from VU0 memory to the scratchpad, where it is transferred to the GIF, as we did in Example 4.

Summary

This example consists of two programs. One is the VU0 program, that uses a transformation matrix and 4 vertices stored in VU0 memory, and transforms all the vertices and stores them elsewhere in VU0 memory. The other is the ‘main’ program and runs on the EE core. It creates a transformation matrix every frame and uploads it to VU0 memory using the DMA. This DMA list also activates the VU0 microprogram. When the microprogram is complete, the results are copied out of VU0 memory into a new DMA list. This is a GS packet that is transferred to the GIF.

Section I - Example 6 – VU1 micromode

In this example, we will use the VU1 instead of the VU0.

The VU1 can dynamically create GS packets and send them to the GIF via a direct connection called PATH1. This can remove some of the workload from the EE core.

Instead of the VU0 calculating the transformed vertices and the EE copying the results to scratchpad RAM, the example runs a VU1 microprogram that creates the GS packet in VU1 RAM. This packet is then sent to the GIF over PATH1.

This example is very similar to Example 5. The main difference is that the VU1 program creates the GS packet on the fly, instead of the EE core creating it on scratchpad RAM.

Example 6 – VU assembler walkthough

This program is very similar to the previous example. This section only covers the changes.

```
UNPACK 4, 4, V4_32, 4, *     ;// Vertices
.float  -100, -100, 0, 1
.float   100, -100, 0, 1
.float  -100,  100, 0, 1
.float   100,  100, 0, 1

GIFpacked PRIM=0X04, REGS={ RGBAQ, XYZ2, XYZ2, XYZ2, XYZ2 }, NLOOP=1, EOP
.EndGif
.word   0x80, 0x80, 0, 0     ; Colour
.EndUnpack
```

This UNPACK instruction previously transferred the source vertices. In this example, it additionally transfers the GIF tag we will use. The GIF tag never changes, so we can assemble it in advance.

After the GIF tag comes the colour, which is the first GS register setting in the GS packet.

The program will build the GS packet in the area directly following this quadword.

```
NOP                                 IADDIU      VI04, VI00, 10
```

This instruction has changed from the previous example. It loads the destination address of the transformed vertices into VI04. This value has been changed from 8 to 10. This is because the GIF tag now lies at address 8, and the colour lies at address 9. The VU program will place the transformed vertices starting at address 10. This will complete the GS packet. [Recall that the GS packet will be comprised of the GIF tag, a polygon colour, and 4 vertex settings.]

```
NOP                                 IADDIU    VI04, VI00, 8   ; Address of GIFtag
NOP[e]                              XGKICK    VI04
```

There are two new instructions here, directly after the conditional branch. The first loads the value 8 into register VI04. This is the address of the GS packet that we want to send to the GIF.

The second is a new instruction, only available on VU1. This is the `XGKICK` instruction, and it sends the GS packet to the GIF via PATH1. The parameter is a register holding the start address of the GS packet.

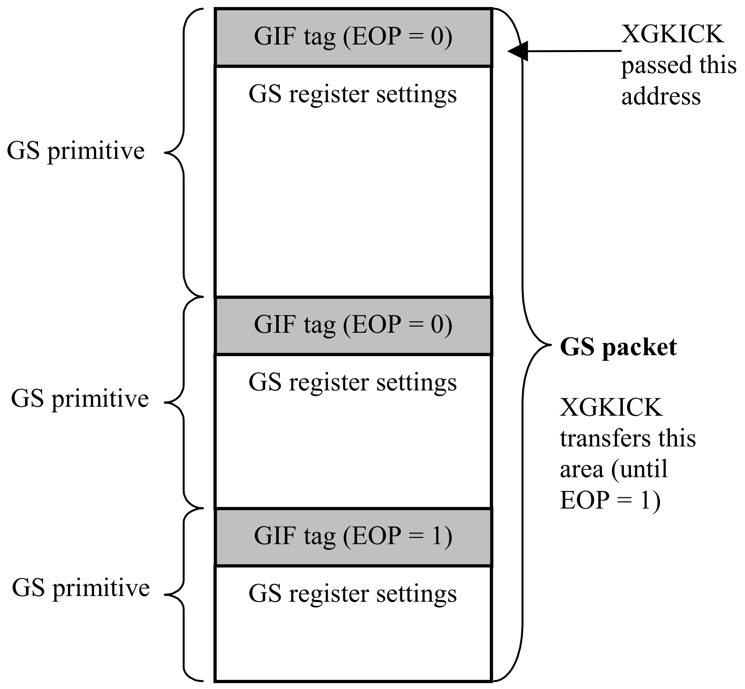
The `[e]` bit is set on the last line, so the microprogram will halt after the XGKICK instruction.

EOP bit revisited

Recall that the EOP bit is used in the GIF tag to prevent PATH switches. When a GS packet is sent from the VU1, it is used to help determine when the XGKICK should stop transferring data.

The XGKICK instruction is passed the address of a GIF tag. It sends this GIF tag to the GIF, and sends the following quadwords as the data. It knows the length of the data following the GIF tag, because this can be determined by the `FLG`, `NREG` and `NLOOP` fields in the GIF tag.

When the GIF tag and data has been transferred by the XGKICK, it checks the EOP bit in the GIF tag. If this was 1, the `XGKICK` finishes the transfer. If EOP is 0, the XGKICK moves onto the next quadword and starts the cycle over again. This way, multiple GS primitives can be sent with one `XGKICK` instruction.



Note 1: In this example, we are sending a single GS packet, which is comprised of a single GS primitive.

Note 2 : XGKICK starts the transfer, but unless a transfer is already in progress, it will not stall the pipeline. Thus it is possible to write programs that create GS packets and send them using `XGKICK`, then continue processing more data while the `XGKICK` transfers the packet in the background.

Example 6 – C program walkthrough

There is little to walkthrough in this new version, because more of the workload is now performed by VU1. The program still creates the transformation matrix, uploads it and activates the program, but no longer has to build the GS packet. There is no other task for it to perform.

Summary

This example changed the VU0 microprogram so that it ran on VU1, and built the GS packet directly in VU1 RAM. The microprogram sends that over PATH1 to the GIF, so the EE core no longer has to perform the task of building and sending the GS packet.

Section J - Example 7 – Field & Frame mode

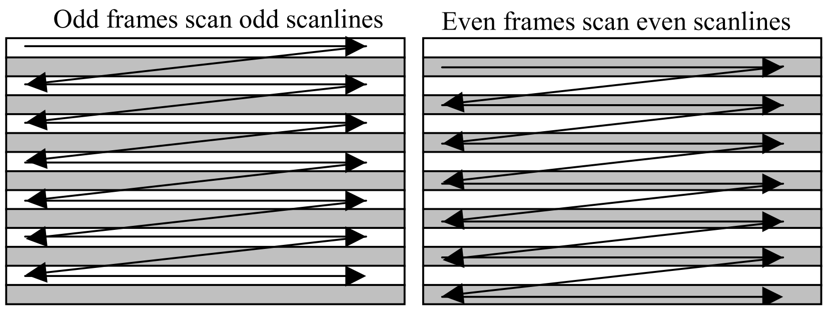
This example makes a very simple change to the code which halves the amount of VRAM required for the display buffers.

Television Fields

If you have never worked with television display output, or have only worked on low-resolution (non-interlaced) games, you will need to know about television fields.

A television has a number of scanlines running left to right that the electron beam travels along drawing the picture. The TV scans the electron beam from left to right, top to bottom, and performs this every 60th of a second (NTSC, as used in Japan and the US) or every 50th of a second (PAL, as used in Europe and Oceania). One complete scan from top to bottom is called a frame.

When the standard was created many years ago, the technology was not sufficiently advanced to scan along every scanline every frame. Instead, the beam alternates between odd and even scanlines every frame. That is, in the first frame only the even scanlines are scanned, then in the next frame the odd scanlines are scanned, and so on.


Odd frames scan odd scanlines
Even frames scan even scanlines

Because the scanlines are drawn in an interleaved fashion, this method is called “interlacing”. All PlayStation 2 games should run in interlace mode.

The set of either odd scanlines or even scanlines is called a *field*.

When the Graphics Synthesiser was initialised in the previous examples, the call used was:

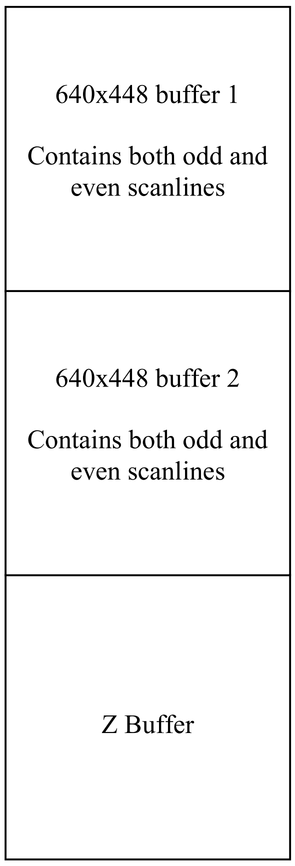
```
sceGsResetGraph(
    0,
    SCE_GS_INTERLACE,
    SCE_GS_NTSC,
    SCE_GS_FIELD);
```

The 2nd parameter specified whether the GS synthesiser should output the graphics in interlace mode. If it is not in interlace mode (`SCE_GS_NOINTERLACE`), then every frame is output at 50/60 Hz to the *even* scanlines only.

The advantage of `SCE_GS_NOINTERLACE` mode is that if the game slows down and therefore takes more than a frame to swap the double buffers, all the user will notice is a frame rate slowdown. This is because the previous buffer will be drawn over the same scanlines, so it looks like the same image is staying on screen longer. This is a moot point however, since all all PlayStation 2 games should run in interlace mode (that is, use both fields of the TV display).

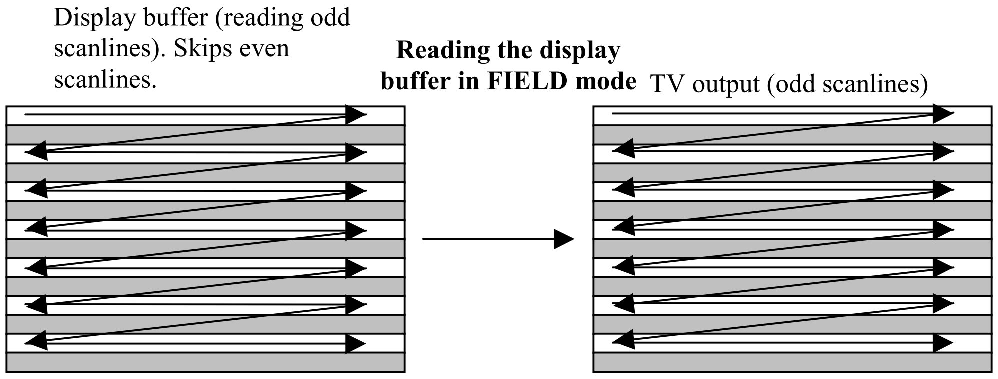
Field/Frame based drawing

If the odd *and* even fields are to be displayed then how is this arranged in the drawing buffer?


640x448 buffer 1
Contains both odd and even scanlines
640x448 buffer 2
Contains both odd and even scanlines
Z Buffer

This is one possible configuration, set by using the `SCE_GS_FIELD` mode, and was used for all previous examples. There are two buffers (for double buffering), and each buffer is 640x448 in size. (The size is arbitrary, 640x448 is a typical buffer size). Each buffer contains both the odd and even scanlines, interleaved. We set the GS to display buffer 1 while drawing to buffer 2, then swap the buffers when drawing is complete.

If the GS is set to `SCE_GS_FIELD` mode (or simply *field mode*), then it will know that the odd and even scanlines of the display buffer are interleaved, and read alternate lines in the display buffer when sending the lines to the TV. Thus the correct image will be output to the screen.


Display buffer (reading odd scanlines). Skips even scanlines.
Reading the display buffer in FIELD mode
TV output (odd scanlines)

(This diagram only displays the buffer reading the odd scanlines. The scanlines are alternated for the even scanlines)

The advantage of this mode is that if it takes us more than a frame to update the drawing buffer, the display hardware will continue to use the current display buffer. It will cycle between sending the odd and even display buffer lines to the TV.

The disadvantage with this method is that it uses a lot of GS VRAM. With only 4MB of VRAM inside the GS, space is at a premium, so any technique that can be used to save space is worth trying.

Frame mode

While giving you the option to use field mode, the designers of the PlayStation 2 realised that most games will run at 50Hz (PAL) or 60Hz (NTSC). This allows some memory saving optimisations.

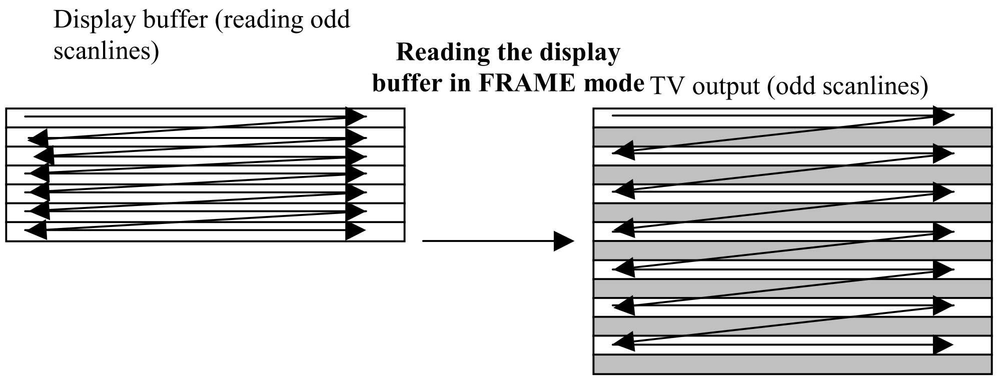
If the game updates the drawing buffer at full frame rate (50/60 Hz) then the program will be swapping between display and drawing buffers every frame. This will mean that half the memory is being wasted. This is because the program is swapping draw/display buffers at the same rate as the frame rate, and therefore one buffer will only ever display odd scanlines while the other buffer only ever displays even scanlines. (This is *only* true if the display buffer is updated and swapped every frame)

Let's say that although buffer 1 is 640x448 high, only the odd lines will ever be displayed in it. And although buffer 2 is also 640x448 high, only the even scanlines will ever be displayed in it. In this case, what is the point of storing the even scanlines in buffer 1, and the odd scanlines in buffer 2? In addition, half of the Z buffer is wasted because the half of the lines written to are never displayed.

The designers of the GS saw that if (and only if) the drawing/display buffers were updated and swapped every frame, then half the GS memory would be wasted. Therefore there is another mode, called *frame mode* that the GS can use to display buffers.

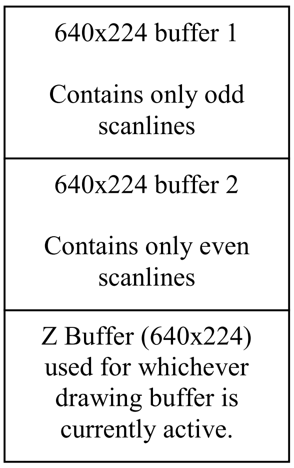
In this mode, the GS does not skip every other line when it reads the display buffer to output the scanlines to the TV. Instead it assumes the buffer holds *only* the odd or even fields, and reads consecutive lines. It still outputs to every other scanline on the TV, of course.

Reading the display buffer in FRAME mode


Display buffer (reading odd scanlines)
TV output (odd scanlines)

(This diagram only displays the buffer reading the odd scanlines. The scanlines are alternated for the even scanlines)

So instead of defining the two buffers to be 640x448 each, it is now possible to define them to be only half the vertical height, since only half of the scanlines will be used. So each buffer is now 640x224 in dimension. One will display the odd scanlines and the other will display the even scanlines. The Z buffer can also afford to be half the size. This technique has halved the memory requirements for the display/draw buffers! The main proviso is that the game must update and swap the buffers at full frame rate (50/60Hz).


640x224 buffer 1
Contains only odd scanlines
640x224 buffer 2
Contains only even scanlines
Z Buffer (640x224) used for whichever drawing buffer is currently active.

Obviously, *frame based drawing* is the best method to use, since it saves so much VRAM. When using this mode, you have to be aware of a few things:

Firstly, the update rate **must** stay at 50Hz (PAL) or 60Hz (NTSC). If your game slows down and you miss swapping the buffers, then what is in the buffers will be out of sync with what the TV is expecting and the screen will display the wrong buffer. This will look like the game has dropped into low-resolution (no-interlace) mode, and is a very ugly effect.

Secondly, each buffer only stores either the odd or even scanlines. Therefore the rasteriser (the part of the GS that actually draws the polygons to VRAM) must render only the odd or even scanlines to the appropriate buffer. But the rasteriser does not know about even and odd scanlines…

To get the rasteriser to render the even scanlines only, it is required that all the Y vertices of any polygons must be scaled to half their size before being transformed to the 2048.0,2048.0 centre of the screen. Normally you would perform some transformation to obtain the Y component around the (0, 0, 0) origin, then add 2048.0 to the X and Y to centre them within GS primitive space. But now, you must halve the Y vertex before adding the 2048.0 value to the Y component.

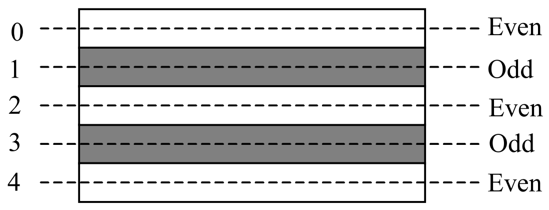
The CameraToScreen Matrix can take care of this, so an extra mathematical operation is not required. One of the parameters to the `sceVu0ViewScreenMatrix` function is the X and Y scale. If you set the Y scale to 0.5, then the Y vertices will be halved before the 2048.0 value is added in the transformation.

Half Pixel Offset

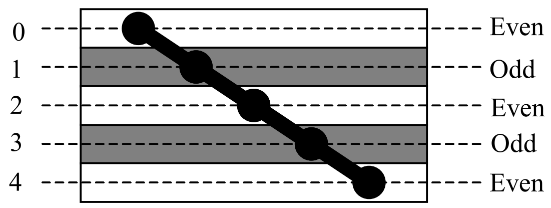
Drawing the odd scanlines requires and extra operation. It is necessary to adjust the Y offset for odd frames.

"Half Pixel Offset" is the term used to describe the change in Y offset used every frame. If the program is running in interlaced mode *and* is rendering using frame based drawing, then Half Pixel Offset correction is required.

To explain why this is needed, here is a diagram of part of the screen display. Each box represents a scanline on the television screen. The display alternated between displaying even and odd frames. Even scanlines are physically positioned on the TV screen between the odd scanlines.



In this example, we wish to draw a line from scanline 0 to scanline 4. For example purposes, let's say the Y drawing offset is currently 0.



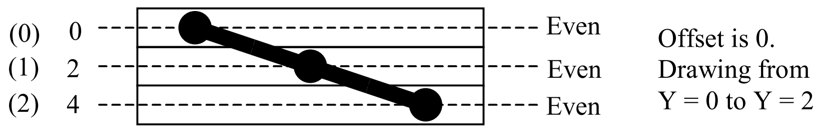
If we were using field based drawing, where even and odd scanlines are interleaved, then the start/end Y coordinates would be 0.0 and 4.0 respectively. This is what we'd expect.

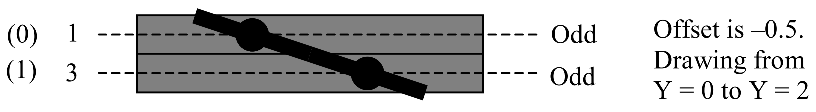
However, using *frame based drawing* where even and odd scanlines are *not* interleaved, we have to specify different Y coordinates for both even and odd frames.

In the even frame buffer, only even scanlines are stored (0, 2, 4, 6, …). Therefore instead of specifying 0 and 4 as the start and endpoints, we must halve the Y coordinates and instead specify 0 and 2. The point sampler in the GS rendering hardware will calculate the intersection of the line and the middle of the three scanlines in the way we would expect.

This renders correctly on the even frames, but not on the odd frames. For the odd frames we want to draw onto scanline 1 and scanline 3. So we still specify 0 and 2 as the Y coordinates but subtract 0.5 from the Y offset before doing so. This way we are effectively drawing the same line from –0.5 to 1.5. Because the GS expects drawing coordinates in a fixed-point format, it is possible to specify fractional coordinates such as this.

The scanline renderer will point sample this line from –0.5 to 1.5 and the intersection with the middle of the frame buffer line will be in the appropriate place.





When even frames are to be drawn, the Y offset must be reset to its original value. When odd frames are drawn, the Y offset must be reset to the original value minus 0.5.

The `sceGsSetHalfOffset` function changes the double buffer structure to the correct offset. The next time the double buffers are swapped, the changes will take effect.

Example 7 of the sample code that uses frame based rendering. The output is identical to Example 6, but half the GS VRAM is used. Here are the important parts of the changes made to Example 6 to get it to run with *frame based* rendering.

Walkthrough

```
#define SCREEN_HEIGHT 224        // NOTE: This is now half the size of the original
```

The screen height is halved, because each buffer now contains half the scanlines (either the odd or even scanlines).

```
    int field;                     // current display field
```

Instead of storing which buffer we are using, we store which field is being drawn to (0 for even, 1 for odd).

```
    sceGsDBuff db, *p_db;                   // Double buffer handler
    p_db = (sceGsDBuff *)(((u_int)&db & 0x0fffffff) | 0x20000000);
```

We use an uncached pointer to the double buffer structure. Part of this structure is transferred by the DMA to the GIF every frame so that the buffers can be swapped. It also contains the Y offset that needs to change every frame. Because the program alters this value, and the value has to be transferred by DMA, we write to uncached memory to avoid having to flush the cache.

```
    sceGsResetGraph(
        0,
        SCE_GS_INTERLACE,
        SCE_GS_NTSC,
        SCE_GS_FRAME);
```

The initialisation of the screen. Notice the last parameter, which is now `SCE_GS_FRAME` instead of `SCE_GS_FIELD`.

```
        field = sceGsSyncV(0);              // Wait for vertical blank
        sceGsSwapDBuff(p_db, !field);       // Swap double buffers
```

The first part of the main loop. The `sceGsSyncV` function returns the current field being displayed (0 is even, 1 is odd). We use this to decide which double buffer to display.

```
            // add half pixel to Y offset address for interlace
        sceGsSetHalfOffset((field&1)?(&p_db->draw0):(&p_db->draw1),
            2048, 2048, 1-field);
```

This changes the double buffer structure so that the field in question has the correct half offset adjusted. The double buffer structures are DMA'd to the GS, so it's necessary to write to supply a pointer to the structures in uncached memory. The second and third parameters are the original screen center in GS primitive space, and the final parameter is a flag stating whether an adjustment for odd or even fields should be made. If this is 0, an offset of 0 is made. If it is 1, then an offset of 0.5 is added. [Note: The function adds 0.5 rather than subtracts it. As long as odd scanlines are consistent with the half offset, there is no visible difference]

```
    sceVu0ViewScreenMatrix(
        CameraToScreenMatrix,
        512.0f,
        1.0f,
        0.5f,
        2048.0f,
        2048.0f,
        1.0f,
        16777215.0f,
        64.0f,
        65536.0f
        );
```

This is the Camera To Screen matrix. Notice that the Y scale has been changed from 1.0 to 0.5. This will squash all polygons down to half their vertical size, although (2048,2048) is still the centre of the screen.

The VU1 microcode was not changed.

Summary

The amount of VRAM used can be halved if the frame rate is kept constant at 50 Hz for PAL and 60 Hz for NTSC.

This is because in an interlaced screen, only the even or odd scanlines are every drawn in one frame.

The adjustments made to the code are:

- The GS initialisation (to use field based drawing).
- The transformation matrix (which scales the polygons down to half their vertical height).
- The Half Pixel Offset (so that the correct adjustment is made for odd scanlines).

Section K – Example 8 – Controllers and Console

In this example, we introduce an easy way to get controller input, and to output text messages to the TV screen.

When experimenting with programs, it's often useful to have some form of rudimentary input and output. You have seen very rudimentary output already – when you call the `printf` function, the output goes to the console.

However, this is very crude, especially if you want to track a variable over time. Printing it to the console every frame will generate a lot of output and clutter the console window. In some cases, it's preferable to output the value to the same place on the screen.

In this example, we use the `libdev` library, which assists with certain debugging functions. One of its functions is to manage a psuedo console window on the TV screen. You can output strings to this pseudo console and they will be displayed on top of your graphics. In this example, we use these functions to display the values of the rotation vector.

The `libpad` library handles the controllers. In a 'real-world' title, controller code is quite sophisticated, because it has to manage different types of controllers, and events like the controller being removed. However this example demonstrates a very simple way to obtain raw button data, which is used to rotate the plane on screen in the X, Y and Z dimensions.

The VU1 code remains the same as the code used in Example 6, so it is not repeated here.

Walkthrough

```
    // Pad buffer
u_long128 padDma[scePadDmaBufferMax] __attribute__((aligned (64)));
u_int paddata;
u_char rdata[32];
```

Many years ago, game controllers were simply a set of microswitches that closed circuit when pressed, and the console could tell if a button was pressed just by looking at the pins.

Today, controllers are more sophisticated. On the PlayStation and PlayStation 2, the each button does not have a wire that is linked directly to the console. Instead, the controller contains a microprocessor, and communicates with the console via a serial link.

When it is plugged in, the controller and application negotiate a mode to work in (for instance, the application can request that the Dual Shock 2 controller locks into analog mode). Then the controller will send packets of data describing the state of its buttons. On a Dual Shock 2, this packet includes which buttons are being pressed, how hard they are being pressed, and the position of the analog controllers. The application may even send data back to the controller, for example vibration control data.

The pad libraries need an area in EE RAM to receive packets from a controller. This is the `padDma` area defined above, and will be passed to the pad libraries when we open that controller port.

The `paddata` and `rdata` variables are used as temporary buffers when interpreting this packet data.

```
    sceSifInitRpc(0);
```

The `sceSifInitRpc` function initialises the Remote Call Procedure library. This library allows the EE to call functions on the IOP, and have values returned. The program will be calling the `sceSifLoadModule` function that uses SIF RPC, so it is necessary that the first task the program performs is to initialise the library.

```
    while(sceSifLoadModule("host0:/usr/local/sce/iop/modules/sio2man.irx",0,NULL)<0){
        printf("Retrying module sio2man\n");
    }
```

```
while(sceSifLoadModule("host0:/usr/local/sce/iop/modules/padman.irx",0,NULL)<0){
    printf("Retrying module padman\n");
}
```

The `sceSifLoadModule` function is used to load a module from disc. A module is an IOP program. These programs run as concurrent threads in the IOP. This `sceSifLoadModule` function will read the module from the disc, load it into IOP RAM and start its execution.

Because the disc may be dirty or scratched, the program may have to retry loading if the module fails to load the first time. This is why the `sceSifLoadModule` is in a while loop.

The two IOP modules that are being loaded are `sio2man.irx` (A manager that handles basic input and output) and `padman.irx` (that handles communication with the controllers). The program needs these two modules loaded before the EE can communicate with the controllers via the IOP.

```
scePadInit(0);                    // Open controller
scePadPortOpen(0, 0, padDma);
```

Once the modules are loaded, it is possible to use them. There is an EE library called `libpad` that uses the SIF RPC to call the controller management functions within `padman.irx`. The first call initialises the `libpad` libraries, which makes sure it can communicate with the `padman.irx` module on the IOP.

The second call opens Port 0, Slot 0. The Port number is either 0 or 1. Port 0 is the left hand controller port, and Port 1 is the right hand controller port. The slot number is only appropriate if a multi-tap (4 controller connection box) is plugged in. Otherwise, it should be left at 0. This program assumes that a Dual Shock 2 controller is plugged into Port 0, the left hand controller port.

The `padDma` argument is a buffer to store packet information from this controller. It arrives from the controller at the IOP, and is transferred by `padman.irx` into the `padDma` area. The data is a maximum of 256 bytes long, but must be aligned to a 16 byte boundary, as it is transferred from the IOP by the DMA controller, which is strict about data alignment.

`Padman.irx` will regularly (i.e. at least every 50th of a second) fill this space with the latest packet data from the controller.

Examining the state of the buttons

To examine the data, the program needs to call the following:

```
    // Read controller
if(scePadRead(0, 0, rdata) > 0){
    paddata = 0xffff ^ ((rdata[2] << 8) | rdata[3]);
} else {
    paddata = 0;
}
```

The `scePadRead` function examines the data that last arrived from port and slot specified by the first two arguments. It assumes that the controller has already been opened by `scePadPortOpen`. It interprets the last packet of data that was received, and puts it into an easy to interpret format in `rdata`, a 32 byte buffer that was allocated previously. If there has been a communication error with the controller, `scePadRead` will return 0 or less.

The exact contents of `rdata` after the `scePadRead` call are specified in the library documentation, but we are only interested in two bytes. These two bytes (`rdata[2]` and `rdata[3]`) store the state of the 16 buttons on the Dual Shock 2 controller. Each bit corresponds to one button on the controller. These bitfields are inverted (1 means button is not pressed, 0 means button is pressed), so the values are inverted to get a more intuitive value.

There are macros to reference each button, so you do not have to remember the bit positions of each button. However, they are named a little unusually. For example, `SCE_PADLup` refers to the “up” button on the directional pad on the Left. `SCE_PADRup` refers to the “up” button on the right of the controller, which is the triangle button. The other buttons are named as you’d expect (`SCE_PADL1`, `SCE_PADR1`, `SCE_PADstart`, etc). However the L3 and R3 buttons (pressed when you push down on the analog sticks) are defined as `SCE_PADi` and `SCE_PADj`.

The example program tests these bits and uses them to adjust the rotation vector.

This is quick and dirty way to get controller input. In general, controllers need more management. There are two reasons for this. Firstly, when a controller is plugged in, it must negotiate with the console and the application. It may take a few milliseconds to become stable. If the application decides to change a controller's mode, then it may take a few milliseconds to adjust to that as well. The game must take into account that it may not always have a controller to read.

Also, the Technical Requirements Checklist states that the program must pause with a message if an invalid controller is inserted, or if a controller is removed during gameplay. A 'real-world' application cannot simply read the incoming packet data and assume that a controller will be plugged in. However, this is fine for our example purposes.

Console output

This program displays the rotation coordinates on the screen. This is very useful for debugging. To achieve this, it uses `libdev`, a library with various functions that assist debugging.

This library can open a pseudo console window on the main screen. This console window can have text characters sent to it, just like the normal console window.

```
sceDevConsInit();   // Initialise screen console output
console = sceDevConsOpen(
  (2048 - SCREEN_WIDTH/2 + 20) << 4,  // top left GS X primitive coord
  (2048 - SCREEN_HEIGHT/2 + 20) << 4, // top left GS Y primitive coord
  80, // Number of chars (X)
  5); // Number of chars (Y)
sceDevConsAttribute(console, 7);    // Output in white (default colours can be
                                    // redefined by sceDevConsSetColor)
```

Here the program initialises the library and opens the console window. Only one pseudo console window can be opened at once. The `sceDevConsOpen` function returns a console handler id, which is used to change attributes of the open console window.

The first two arguments to `sceDevConsOpen` are the coordinates of the top left corner of the console window, in GS primitive space. Since we know that (using the standard libraries) the centre is (2048, 2048), and we know the screen width/height, it is simple to calculate the primitive coordinates of the top left corner of the screen. However, we place the console window at a position 20 pixels in from the top edge and left edge, because some TVs crop a few lines off our display.

The last two arguments are the size of the console window, measured in characters. With the current release (V2.1.2 at time of writing), the console window can be a maximum of 80 by 40 characters.

The pseudo console has 8 pre-defined colours that it can output the text in, numbered 0 to 7. The default output colour is colour 0 (which is black). You can redefine the default colours using the `sceDevConsSetColor` function. We need to change the current output colour from black to something else, otherwise we won't be able to see the text against the black background.

The `sceDevConsAttribute` function changes the colour of any text that will be subsequently sent to the console. We set it to 7, which is white by default.

Sending text to the pseudo console

```
sceDevConsClear(console);
sceDevConsPrintf(console,            // Output rotation values to screen
    "X rotation : %2.4f  Y rotation : %2.4f  Z rotation : %2.4f",
    rotation[0], rotation[1], rotation[2]);
sceDevConsDraw(console);
```

In the main loop, we want to output the values of the rotation vector. If we add text to the console using `sceDevConsPrintf` then the output will be added onto the end of what is currently there, as the pseudo console emulates a normal console. If it fills or overflows the console window, then the console window will ‘scroll’ to fit the new text in, just like a normal console.

We want to display the same text at the same place every frame. Therefore we call `sceDevConsClear`, which will clear the pseudo console window and reset the cursor to the top left. If we did not call this function, then the console area would be filled up with output, would scroll every frame and therefore be difficult to read.

The `sceDevConsPrintf` function is used just like the normal `printf`, except the console id is passed in as the first argument.

Finally, to draw the contents of the console, the function `sceDevConsDraw` is called. This builds the packets in an internal buffer and sends them via PATH3 to the GIF. When it returns, the console output will be in the drawing buffer.

Summary

This example covers very rudimentary ways of getting text output onto the screen and getting rudimentary input from a controller (both useful for debugging). Neither method would be used in production code, but is useful for the beginner.

Section L – More about the system

Now that we have completed the examples, you should understand the following concepts:

- An overview of the system components (EE, IOP, SPU2 and GS).
- A basic understanding of the major EE components (EE core, DMAC, VIF0/VIF1, VU0/VU1, GIF)
- The ability to initiate DMA transfers to the GIF, VIF0 or VIF1, using either Normal Mode or Chain Mode (DMA tags).
- The ability to create GS packets in REGLIST mode and PACKED mode.
- A general understanding of the way a VU works, including how it performs calculations and runs programs.
- The ability to upload VU microprograms and run them.
- An understanding of Frame and Field modes, and how field mode save a lot of VRAM.

These are the foundations of becoming a good EE programmer. It is essential that you have a good understanding of the DMA, the GIF tags and the GS. VU microprograms are essential to tapping the power of the system.

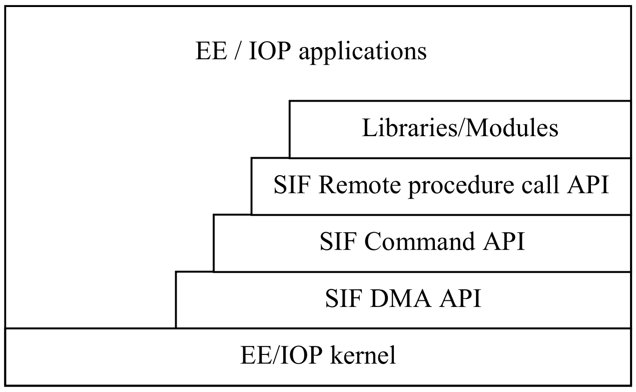
There are some loose ends that need to be covered, but didn't fit into the examples nicely. These will be covered in this section. We will only touch briefly on some areas – this is to help you get a better understanding of how the system works rather than to serve as a complete tutorial.

SIF and the SIF libraries

Recall that the IOP manages all the external peripherals like the memory cards, controllers and CD/DVD. Therefore it is necessary for the EE core to communicate with the IOP. This is performed over the Sub-bus InterFace (SIF). This is a set of 3 DMA channels that can send or receive data to/from the IOP.

SCE has designed a library and communications protocol to handle EE↔IOP data transfers over the SIF. The IOP operating system runs this protocol. Be careful when programming SIF DMA transfers, as you may interfere with the operation of the SIF libraries.

The SCE libraries that manage the SIF are in 3 layers. The layers and interfaces are essentially identical on both the IOP and the EE:



The SIF DMA API is transfers raw pieces of data between the EE and IOP.

The SIF Command API is used on one processor to call functions on the other processor (but no result can be returned).

The SIF Remote Procedure Call API (SIF RPC) is used on one processor to call functions on the other processor *and* return a result.

The EE cannot communicate directly with external peripherals, so many of the SCE EE libraries, such as the memory card, controller and CD/DVD handlers use a combination of the SIF DMA/Command/RPC libraries to communicate with the IOP kernel functions that actually do the work.

For example, when you call a memory card function such as `sceMcOpen()`, this uses the SIF RPC API to make a call to an IOP kernel function which performs the actual operation. Because it would be wasteful for the EE to wait while the IOP calls the function, the `sceMcOpen()` call is asynchronous, and returns immediately. The EE program can continue while the IOP function executes.

The EE can check whether the function has completed by calling `sceMcSync()`. This will return a value stating whether it has received a reply back from the IOP function yet.

EE and IOP kernel replacement

The IOP and EE run custom kernels that are built-in to the PlayStation 2. However it is possible for your application to temporarily replace these kernels with newer ones (supplied by SCE).

On the TOOL, this is achieved by changing the flash ROM in the development kit. This is performed through the '`dsflash`' software, which uploads the new kernels. See the section headed "Changing the flash ROM in the TOOL" in Section A for more details.

It is not possible to flash the ROM on a consumer model, but it is still possible to temporarily upgrade it for the life of your application. For consumer PlayStation 2 applications, the ROMs are kept in an '`.IMG`' file on the CD/DVD, and a function is used when the program starts to load the new kernels. The process generally looks like this (this is EE core code):

```
while (!sceSifRebootIop("cdrom:\\IOPRP20.IMG")); // Load new kernel from disc.
while (!sceSifSyncIop());     // Synchronise with the IOP.

sceCdInit(SCECdINIT);  // Initialises the CD functions.
sceCdMmode(SCECdCD);   // This function selects whether CDs or DVDs are used.

sceSifInitRpc(0);      // this initialises the SIF RPC library.
```

This code is not necessary if you are developing on the TOOL, as the correct flash ROM should already have been installed.

The "`.IMG`" files are stored in `/usr/local/sce/iop/modules/`. The name of the `IMG` file is usually `IOPRP`*xx*`.IMG`, where *xx* is the library release (16 for library release 1.6, 21 for library release 2.1, etc).

Integer Types

It's worth noting that in C/C++:

```
byte           8  bits
short          16 bits
int            32 bits
u_long/long    64 bits
u_long128      128 bits
```

The letter 'L' at the end of a value denotes it as a 64 bit number. (e.g. 0x12345678ABCDEF00**L**)

Alignment

The EE core is very strict about alignment (as are all MIPS processors). You can only load data of a specific size using a single instruction if it is properly aligned.

Bytes	8 bits	alignment does not matter
Half words	16 bits	Lowest bit of address must be 0
Words	32 bits	Lowest 2 bits of address must be 0
Double words	64 bits	Lowest 3 bits of address must be 0
Quad words	128 bits	Lowest 4 bits of address *are considered to be* 0

In all cases except the last, an address alignment exception is generated if the alignment is not correct. In the last case, a quadword load instruction will not cause an address alignment exception if the lower 4 bits are not 0, however the lowest 4 bits will be considered by the memory controller to zero, so you may get unexpected results if they are not.

In order to load data that is not aligned properly, more than one instruction is needed. If you do not align your data in memory, you may be generating more code than necessary.

Alignment is also very important for other parts of the EE. For example, DMA tags must be quadword aligned, and VU microprograms must be aligned on a double word boundary (64 bits).

DMA speed is also affected. You can get a 30% DMA speed increase by aligning your DMA data onto a 128 byte boundary. (Note: This DMA data referred to *includes* DMA tags if you are transferring them [TTE=1], otherwise it does not).

Floating point

It is highly recommended that all references to variables of type '`double`' are removed, and explicitly defined floating point numbers are appended with the letter 'f', e.g. `1.234f`.

Normally the C language will assume that a floating point number such as `1.234` is a double precision number (64-bits). Although the compiler can handle doubles, the code will run much faster if the number is stored as a `float` (single precision, 32 bits), which is the natural storage and mathematical manipulation unit in the Emotion Engine (EE) and the VUs.

When '`f`' is placed at the end of a floating point number in a C file, it is interpreted by the compiler as single precision, not double precision. Using '`f`' at the end of floating point numbers creates code naturally suited to the EE and will significantly increase the speed of standard compiled code.

The PS2 architecture only supports floating point at single precision (32 bits for a single floating point number). However, C compilers by default support floating point with double precision. If you use a floating point number in your code, it will be stored as double precision (64 bits) unless you specify '`f`' at the end of the number.

For example:

```
        c = myFloatVal * 1.5;
```

Here '`c`' and '`myFloatVal`' are defined as a `float`. However the '1.5' is stored internally as double precision, so the resulting operations will be equivalent to:

```
        double temp = floatToDouble(myFloatVal);
        temp = doublePrecisionMultiply (temp, 1.5);
        c = doubleToFloat( temp );
```

All these operations must be done in software as the hardware does not handle double precision numbers. These operations are very slow. However, if the 1.5 was stored as single precision (`1.5f`), then code segment converts to:

```
        c = multiply(myFloatVal, 1.5f);
```

The multiply is a hardware single precision floating point multiply, so obviously this way is a lot faster. If **any** double precision arithmetic is used, the compiler will use software to process it, which is **very** slow in comparison to hardware floating point calculations.

Additionally, make sure that you use the single precision versions of mathematical functions (replace `sin()` with `fsin()`, etc).

short-double

There is a GCC compiler option (`-fshort-double`) that will automatically set the size of a double to the same size as a float (i.e. 32 bits). If you use this compiler option, then any values encountered are considered to be single precision, so you do not have to specify '`f`' at the end of each of your numbers.

However, `printf` (and all `printf` related functions like `sprintf`) always expects a 64 bit double in the format string, and so this may cause problems if you are using `printf`. The solution is to write your own version of `printf` or simply don't use it.

To check if you have removed all double precision references in your code, get the linker to generate a map file and search it for functions like `dptofp` and `fptodp` (These convert between double and single precision). If the linker is adding these functions in, then there must still be an element of your code that is not completely double precision free.

Packet libraries

If you examine the various pieces of sample code in `/usr/local/sce/ee/sample/`, you will see that some of them use a library called `libpkt`, or the Packet Library. This library allows you to construct DMA, VIF and GIF packets, and manages alignment and setting the correct fields in the tags.

It is the author's opinion that you do *NOT* invest time learning and using these libraries. They exist to allow speedy development, but are far from efficient. For a new programmer, your time is better spent learning how the DMA, VIF and GIF work at the register level. The result will be that you'll have a much better understanding of how the DMA/VIF/GIF work, and be able to debug code faster.

DMA Debug library

Invariably, when you are experimenting with building your own DMA, VIF and GIF lists, things will go wrong. For this reason, there is a library on the website (`DMAdebug097.tgz` is the latest version at time of writing) that can disassemble DMA, VIF and GIF lists, and point out errors and possible errors in the lists. Full instructions are included in the package.

Conclusion

Now that you have seen examples of how to use the various components of the EE, you can begin to write simple programs that use these components. There is a lot of material to learn, but the learning curve is fairly steep and you should advance fairly quickly.

It is fairly simple to write a PlayStation 2 program, as you have seen here. And it is obvious that the EE core, VU0, VU1 and the GS are extremely powerful. But it is not obvious how to design a PlayStation 2 program that uses these components to their fullest capabilities.

For this reason, a number of strategies to write PlayStation 2 programs are covered in the second part of this tutorial series.

One of the SCEE's Technology Group's aims is to help you get the best performance from your programs, to truly do justice to the massive power of the PlayStation 2.

Please contact us at `ps2_support@scee.net` (SCEE region developers) or `ps2_support@playstation.sony.com` (SCEA region developers) for advice about designing your PlayStation 2 applications.

Glossary

CD/DVD	Compact Disc/Digital Versatile Disc. The storage medium for PlayStation 2 applications. Applications come on either a CD or a DVD disc. A CD can store about 650 MB, a DVD around 4 GB.
Debug Station	Identical to a normal PlayStation 2, except that it has no copy protection on gold discs. It can play games on gold disc from all three regions, but can only play final games (those with a blue underside pressed by Sony) from its own region.
TOOL	Development Kit. The big black PlayStation 2 used to develop applications. A TOOL has 4 times the EE RAM and 4 times the IOP RAM of a consumer PlayStation 2.
DMA	Direct Memory Access. The mechanism used to transfer data between various sub-systems. In general, a source address, destination sub-system and data size is selected, and the data is automatically transferred to the destination by the DMA Controller (DMAC).
DMAC	DMA Controller. See DMA.
DMA List	A chain of DMA Tags that comprise a single DMA channel data transfer.
DMA Tag	An arbitrarily placed quadword aligned long word (64 bits), that is interpreted by the DMA Controller, and specifies the address and length of data to transfer.
EE	Emotion Engine. The chip that contains a MIPS core, the VUs, VIFs, SPR, DMAC, memory controller, SPR, IPU, GIF and SIF.
GIF	Graphics synthesiser InterFace. The interface between the EE and the GS. VU1, VIF1 and the EE core all have access to the GIF.
GS	Graphics Synthesiser. The chip that draws the polygons and contains the frame buffer and Z buffer.
IOP	Input/Output Processor. The processor that handles all interfaces with peripherals (sound, iLink, CD/DVD, USB, controllers, memory cards), and is also connected to the EE (via the SIF).
IPU	Image Processing Unit. The unit that can perform Discreet Cosine Transforms (DCTs) on portions of images, and perform colour space conversions. Used for processing MPEG and JPEG files
MFIFO	Memory First In First Out. A mechanism for transferring data between one peripheral and another via EE memory (e.g. input SIF to output VIF1), in such a way that the transfer speed of the source does not exceed the transfer speed of the drain.
QWORD	Quad Word. A word that is 16 bytes wide (or 4 words, hence the name). A total of 128 bits. The DMA transfers data in units of one word for the SIF channels, and one qword for all other channels.
SIF	Sub-bus InterFace. The interface used to transfer data between the IOP and the EE. The SIF has 3 DMA channels used for EE<-> IOP communication.
SPR	Scratch Pad Ram – 16KB of RAM embedded on the EE. The start of it is located at address 0x70000000. Because it is on the core, it only takes one cycle to access it, so it is very fast, and useful for storing temporary structures.
VIF	Vector unit InferFace. The interface used to transfer data to a VU’s memory. Each Vector Unit has a VIF. A VIF can perform operations on the data that passes through it (via DMA), and also activate VU programs.

VLIW	Very Long Instruction Word. Describes the opcode architecture of the VUs. In this architecture, a single instruction word contains two instructions, one for the floating point units and the other for the integer unit.
VPU	Vector Processing Unit. See VU.
VU	Vector Unit. A parallel processor embedded on the EE (of which there are two, VU0 and VU1). Each has their own memory for data and code (4KB + 4KB for VU0 and 16Kb+16KB for VU1).

Appendix I – Example source code listings

The full source code to each example is included here. Each example builds on previous examples. Significant changes from one example to the next are highlighted in bold.

Example 1

```
// Example 1 - Puts 2 triangles onto the screen. Uses REGLIST mode & Normal Mode DMA.

#include <eekernel.h>
#include <stdlib.h>
#include <stdio.h>
#include <eeregs.h>
#include <libgraph.h>
#include <libdma.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 448
#define SPR 0x70000000

int main(void){
    int buffer = 0;                 // current display buffer
    sceGsDBuff db;                  // Double buffer handler
    sceDmaChan *dmaGif;             // GIF DMA handler
    u_long *spr;                    // Pointer to scratch pad ram

    const u_long myGIFTag[2] = {
        SCE_GIF_SET_TAG(
            2,      // NLOOP
            1,      // EOP
            0,      // PRE
            0,      // PRIM
            1,      // FLG
            5       // NREG
        ),
        0x0000000000055510L
    };

        // Screen and basic system setup starts here

    sceGsResetPath();
    sceDmaReset(1);

    sceGsResetGraph(
        0,
        SCE_GS_INTERLACE,
        SCE_GS_NTSC,
        SCE_GS_FIELD);

    sceGsSetDefDBuff(
        &db,
        SCE_GS_PSMCT32,
        SCREEN_WIDTH,
        SCREEN_HEIGHT,
        SCE_GS_ZALWAYS,
        SCE_GS_PSMZ24,
        SCE_GS_CLEAR);

    FlushCache(0);

    dmaGif = sceDmaGetChan(SCE_DMA_GIF);

    spr = (u_long *)SPR;

    *(spr++) = myGIFTag[0];
    *(spr++) = myGIFTag[1];

    *(spr++) = SCE_GS_SET_PRIM(
                SCE_GS_PRIM_TRI,    // PRIM (Primitive type)
                0,                  // IIP  (Gouraud)
                0,                  // TME  (Textured)
                0,                  // FGE  (Fogging)
                0,                  // ABE  (Alpha Blending)
                0,                  // AA1  (Anti-Aliasing)
                0,                  // FST  (Use ST for texture coords)
                0,                  // CTXT (Context)
                0);                 // FIX  (Fragment control)
    *(spr++) = SCE_GS_SET_RGBAQ(0x80, 0x00, 0x00, 0x00, 0);
    *(spr++) = SCE_GS_SET_XYZ((2048 - 40) << 4, ((2048 + 40) << 4), 0);
    *(spr++) = SCE_GS_SET_XYZ((2048 + 40) << 4, ((2048 + 40) << 4), 0);
    *(spr++) = SCE_GS_SET_XYZ((2048     ) << 4, ((2048 - 40) << 4), 0);

    *(spr++) = SCE_GS_SET_PRIM(
                SCE_GS_PRIM_TRI,    // PRIM (Primitive type)
                0,                  // IIP  (Gouraud)
                0,                  // TME  (Textured)
                0,                  // FGE  (Fogging)
                0,                  // ABE  (Alpha Blending)
                0,                  // AA1  (Anti-Aliasing)
                0,                  // FST  (Use ST for texture coords)
                0,                  // CTXT (Context)
                0);                 // FIX  (Fragment control)
    *(spr++) = SCE_GS_SET_RGBAQ(0x00, 0x80, 0x00, 0x00, 0);
    *(spr++) = SCE_GS_SET_XYZ((2048 - 40) << 4, ((2048 - 80) << 4), 0);
    *(spr++) = SCE_GS_SET_XYZ((2048 + 40) << 4, ((2048 - 80) << 4), 0);
    *(spr++) = SCE_GS_SET_XYZ((2048     ) << 4, ((2048 -160) << 4), 0);


    asm("sync.l");

    while (1) {
        sceGsSyncV(0);              // Wait for vertical blank
        buffer = 1 - buffer;
        sceGsSwapDBuff(&db, buffer);        // Swap double buffers

        sceDmaSendN(dmaGif, (u_long128*)(SPR | 0x80000000), 6);
    }

    return 0;
}
```

Example 2

```
// Example 2 - DMA tags and PACKED mode

#include <eekernel.h>
#include <stdlib.h>
#include <stdio.h>
#include <eeregs.h>
#include <libgraph.h>
#include <libdma.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 448
#define SPR 0x70000000

u_long bluetriangle[8] __attribute__ ((aligned(16))) = {
    0x0000000000000000L, 0x0000000000000080L,       // colour
    ((2048L - 40) << 4) | ( ((2048L + 40) << 4) << 32 ), 0, // XY,Z
    ((2048L + 40) << 4) | ( ((2048L + 40) << 4) << 32 ), 0, // XY,Z
    ((2048L     ) << 4) | ( ((2048L - 40) << 4) << 32 ), 0  // XY,Z
};

u_long green[2] __attribute__ ((aligned(16))) = {
    0x0000008000000000L, 0x0000000000000000L        // colour
};

const u_long myGIFTag[2] __attribute__ ((aligned(16))) = {
        SCE_GIF_SET_TAG(
            2,      // NLOOP
            1,      // EOP
            1,      // PRE
            SCE_GS_SET_PRIM(
                SCE_GS_PRIM_TRI,    // PRIM (Primitive type)
                0,                  // IIP  (Gouraud)
                0,                  // TME  (Textured)
                0,                  // FGE  (Fogging)
                0,                  // ABE  (Alpha Blending)
                0,                  // AA1  (Anti-Aliasing)
                0,                  // FST  (Use ST for texture coords)
                0,                  // CTXT (Context)
                0),                 // FIX  (Fragment control)
            0,      // FLG
            4       // NREG
        ),
        0x000000000005551L
    };

int main(void){
    int buffer;
    sceGsDBuff db;                  // Double buffer handler
    sceDmaChan *dmaGif;             // GIF DMA handler
    u_long *spr;                    // Pointer to scratch pad ram
    u_long addr;

        // Screen and basic system setup starts here

    sceGsResetPath();
    sceDmaReset(1);

    sceGsResetGraph(
        0,
        SCE_GS_INTERLACE,
        SCE_GS_NTSC,
        SCE_GS_FIELD);

    sceGsSetDefDBuff(
        &db,
        SCE_GS_PSMCT32,
        SCREEN_WIDTH,
        SCREEN_HEIGHT,
        SCE_GS_ZALWAYS,
        SCE_GS_PSMZ24,
        SCE_GS_CLEAR);

    FlushCache(0);

    dmaGif = sceDmaGetChan(SCE_DMA_GIF);

    spr = (u_long *)SPR;

        // Create REF DMA Tag for GIFTag
    addr = (u_long)&myGIFTag;
    *(spr++) = (addr << 32) | (3L << 28) | 1L;      // REF DMA tag
    *(spr++) = 0;     // padding

            // Triangle 1

        // Create REF DMA Tag for blue triangle
    addr = (u_long)&bluetriangle;
    *(spr++) = (addr << 32) | (3L << 28) | 4L;      // REF DMA tag
    *(spr++) = 0;     // padding

            // Triangle 2

        // Create REF DMA Tag for green colour
    addr = (u_long)&green;
    *(spr++) = (addr << 32) | (3L << 28) | 1L;      // REF DMA tag
    *(spr++) = 0;     // padding

        // Create CNT DMA Tag for triangle vertex data
    *(spr++) = (1L << 28) | 3L;      // CNT DMA tag
    *(spr++) = 0;     // padding

    *(spr++) = ((2048L - 40) << 4) | ( ((2048L - 80) << 4) << 32 ); // X and Y
    *(spr++) = 0;     // Z
    *(spr++) = ((2048L + 40) << 4) | ( ((2048L - 80) << 4) << 32 ); // X and Y
    *(spr++) = 0;     // Z
    *(spr++) = ((2048L     ) << 4) | ( ((2048L -160) << 4) << 32 ); // X and Y
    *(spr++) = 0;     // Z

        // Create END DMA Tag for triangle vertex data
    *(spr++) = (7L << 28) | 0L;      // END DMA tag
    *(spr++) = 0;     // padding

    asm("sync.l");
    while (1) {
        sceGsSyncV(0);              // Wait for vertical blank
        buffer = 1 - buffer;
        sceGsSwapDBuff(&db, buffer);        // Swap double buffers

        sceDmaSend(dmaGif, (u_long128*)(SPR | 0x80000000));
    }

    return 0;
}
```

Example 3

```
#include <eekernel.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <eeregs.h>
#include <libgraph.h>
#include <libdma.h>
#include <libvu0.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 448
#define SPR 0x70000000
#define PI (3.141592f)
#define SETVECTOR(v,x,y,z,w)  ((v)[0]=x,(v)[1]=y,(v)[2]=z,(v)[3]=w)

sceVu0FVECTOR vertices[4] = {       // vertices of the square
    { -100.0f, -100.0f, 0.0f, 1.0f },
    {  100.0f, -100.0f, 0.0f, 1.0f },
    { -100.0f,  100.0f, 0.0f, 1.0f },
    {  100.0f,  100.0f, 0.0f, 1.0f }
};

u_long polyColour[2] __attribute__ ((aligned(16))) = {
    0x0000008000000000L, 0x0000000000000000L
};

const u_long squareGIFTag[2] = {
    SCE_GIF_SET_TAG(
        1,          // NLOOP (Leave at 0, libraries fill this in)
        1,          // EOP (End Of Packet)
        1,          // PRE (PRIM field Enable)
        SCE_GS_SET_PRIM(    // PRIM field if PRE = 1
            SCE_GS_PRIM_TRISTRIP,   // PRIM (Type of drawing primitive)
            0,                      // IIP (Flat Shading/Gouraud Shading)
            0,                      // TME (Texture Mapping Enabled)
            0,                      // FGE (Fogging Enabled)
            0,                      // ABE (Alpha Blending Enabled)
            0,                      // AA1 (1 Pass Anti-Aliasing Enabled)
            0,                      // FST (Method of specifying Texture Coords)
            0,                      // CTXT (Context)
            0                       // FIX (Fragment Value Control)
            ),
        0,          // FLG (Data format, 0 = PACKED, 1 = REGLIST, 2 = IMAGE)
        5),         // NREG (Number of registers)
    0x55551L
};

sceVu0FMATRIX                       // Vertex transformation matrices
    LocalToWorldMatrix,
    LocalToScreenMatrix,
    WorldToScreenMatrix,
    CameraToScreenMatrix,
    WorldToCameraMatrix;

sceVu0FVECTOR rotation;             // Holds the vector that is used to create
                                    // the rotation matrix
sceVu0FVECTOR translation;          // The translation portion of the rotation matrix

void init(void);

int main(void){
```

```
int buffer = 0;                  // current display buffer
sceGsDBuff db;                   // Double buffer handler
sceDmaChan *dmaGif;              // GIF DMA handler
u_long *spr;                     // Pointer to scratch pad ram
float delta_rot = 1.0f * PI / 180.0f;       // change in rotation

    // Screen and basic system setup starts here

sceGsResetPath();
sceDmaReset(1);

sceGsResetGraph(
    0,
    SCE_GS_INTERLACE,
    SCE_GS_NTSC,
    SCE_GS_FIELD);

sceGsSetDefDBuff(
    &db,
    SCE_GS_PSMCT32,
    SCREEN_WIDTH,
    SCREEN_HEIGHT,
    SCE_GS_ZALWAYS,
    SCE_GS_PSMZ24,
    SCE_GS_CLEAR);

init();

FlushCache(0);

dmaGif = sceDmaGetChan(SCE_DMA_GIF);

SETVECTOR(translation,  0.0f, 0.0f, 0.0f, 0.0f);
SETVECTOR(rotation,     0.0f, 0.0f, 0.0f, 0.0f);

while (1) {
    int i;
    sceGsSyncV(0);               // Wait for vertical blank
    buffer = 1 - buffer;
    sceGsSwapDBuff(&db, buffer);        // Swap double buffers

        // -------------------------------------------------------
        // Calculate the rotation and Local To Screen
        // -------------------------------------------------------

            // Rotate the planes around the axes.

    rotation[0] += delta_rot     ; if(rotation[0] > PI) rotation[0] -= 2.0f * PI;
    rotation[1] += delta_rot/2.0f; if(rotation[1] > PI) rotation[1] -= 2.0f * PI;
    rotation[2] += delta_rot/3.0f; if(rotation[2] > PI) rotation[2] -= 2.0f * PI;

            // Create Local to World matrix
    {
        sceVu0FMATRIX work;
        sceVu0UnitMatrix(work);
        sceVu0RotMatrix(work, work, rotation);
        sceVu0TransMatrix(LocalToWorldMatrix, work, translation);
    }
            // Create the final Local->Screen matrix
    sceVu0MulMatrix(LocalToScreenMatrix, WorldToScreenMatrix, LocalToWorldMatrix);

    spr = (u_long *)SPR;
    *(spr++) = squareGIFTag[0];
    *(spr++) = squareGIFTag[1];
    *(spr++) = polyColour[0];
    *(spr++) = polyColour[1];

        // -------------------------------------------------------
        // Perform vertex transformations
        // -------------------------------------------------------

    for(i = 0; i < 4; i++) {            // Transform each vertex
        sceVu0FVECTOR V;
        float q;
        int j;

        sceVu0ApplyMatrix(V, LocalToScreenMatrix, vertices[i]);
```

```
            q = 1.0f / V[3];                // q = 1/V.w

            for(j = 0; j < 4; j++)
                V[j] *= q;

            /* V now contains the correct screen coordinates in floating
            point format. They must be converted to integer fixed point format
            before they can be sent to the GS. */

            V[3] = 0.0f;                        // Clear the 'W' value in XYZW.

            sceVu0FTOI4Vector((void *)spr, V);      // Convert and save V
            spr += 2;
        }

        asm("sync.l");

        sceDmaSendN(dmaGif, (u_long128*)(SPR | 0x80000000), 6);
    }

    return 0;
}

// ===============================================================================
// Creates a World To Screen matrix
// ===============================================================================

void create_camera(
        sceVu0FMATRIX WorldToCameraMatrix,
        sceVu0FVECTOR camera_rot,
        sceVu0FVECTOR camera_p,
        sceVu0FVECTOR camera_zd,
        sceVu0FVECTOR camera_yd) {

    sceVu0FMATRIX work;
    sceVu0FVECTOR camera_p2, camera_zd2, camera_yd2;

    sceVu0UnitMatrix(work);
    sceVu0RotMatrixX(work,work,camera_rot[0]);
    sceVu0RotMatrixY(work,work,camera_rot[1]);
    sceVu0ApplyMatrix(camera_zd2, work, camera_zd);
    sceVu0ApplyMatrix(camera_yd2, work, camera_yd);
    sceVu0ApplyMatrix(camera_p2, work, camera_p);
    sceVu0CameraMatrix(WorldToCameraMatrix, camera_p2, camera_zd2, camera_yd2);
}

// ===============================================================================
// Initialise the system and program variables
// ===============================================================================
void init(void) {

            // Camera position and orientation

    sceVu0FVECTOR camera_p  = { 0.0f,  0.0f, -400.0f, 1.0f };
    sceVu0FVECTOR camera_zd = { 0.0f,  0.0f, 1.0f, 1.0f };
    sceVu0FVECTOR camera_yd = { 0.0f, -1.0f, 0.0f, 1.0f };
    sceVu0FVECTOR camera_rot = { 0.0f, 0.0f, 0.0f, 1.0f };

    create_camera(WorldToCameraMatrix, camera_rot, camera_p, camera_zd, camera_yd);

    sceVu0ViewScreenMatrix(
        CameraToScreenMatrix,
        512.0f,
        1.0f,
        1.0f,
        2048.0f,
        2048.0f,
        1.0f,
        16777215.0f,
        64.0f,
        65536.0f
        );

    sceVu0MulMatrix(WorldToScreenMatrix, CameraToScreenMatrix, WorldToCameraMatrix);
}
```

Example 4 – C source

```
#include <eekernel.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <eeregs.h>
#include <libgraph.h>
#include <libdma.h>
#include <libvu0.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 448
#define SPR 0x70000000
#define PI (3.141592f)
#define SETVECTOR(v,x,y,z,w) ((v)[0]=x,(v)[1]=y,(v)[2]=z,(v)[3]=w)

extern u_long Vertices __attribute__((section(".vudata")));
extern u_long128 myVU0microprogram __attribute__((section(".vudata")));
extern u_long128 endOfProgram __attribute__((section(".vudata")));

u_long polyColour[2] __attribute__ ((aligned(16))) = {
    0x0000008000000000L, 0x0000000000000080L
};

const u_long squareGIFTag[2] = {
    SCE_GIF_SET_TAG(
        1,          // NLOOP (Leave at 0, libraries fill this in)
        1,          // EOP (End Of Packet)
        1,          // PRE (PRIM field Enable)
        SCE_GS_SET_PRIM(    // PRIM field if PRE = 1
            SCE_GS_PRIM_TRISTRIP,   // PRIM (Type of drawing primitive)
            0,                      // IIP (Flat Shading/Gouraud Shading)
            0,                      // TME (Texture Mapping Enabled)
            0,                      // FGE (Fogging Enabled)
            0,                      // ABE (Alpha Blending Enabled)
            0,                      // AA1 (1 Pass Anti-Aliasing Enabled)
            0,                      // FST (Method of specifying Texture Coords)
            0,                      // CTXT (Context)
            0                       // FIX (Fragment Value Control)
            ),
        0,          // FLG (Data format, 0 = PACKED, 1 = REGLIST, 2 = IMAGE)
        5),         // NREG (Number of registers)
    0x55551L
};

sceVu0FMATRIX                       // Vertex transformation matrices
    LocalToWorldMatrix,
    WorldToScreenMatrix,
    CameraToScreenMatrix,
    WorldToCameraMatrix;

sceVu0FVECTOR rotation;             // Holds the vector that is used to create
                                    // the rotation matrix
sceVu0FVECTOR translation;          // The translation portion of the rotation matrix

void init(void);
void copyAlignedQWords(u_long128 *dest, u_long128 *src, int numQWords);

int main(void){
    int buffer = 0;                 // current display buffer
    sceGsDBuff db;                  // Double buffer handler
    sceDmaChan *dmaGif;             // GIF DMA handler
    u_long *spr;                    // Pointer to scratch pad ram
    float delta_rot = 1.0f * PI / 180.0f;       // change in rotation

    sceGsResetPath();
    sceDmaReset(1);

    sceGsResetGraph(
        0,
        SCE_GS_INTERLACE,
        SCE_GS_NTSC,
        SCE_GS_FIELD);
    sceGsSetDefDBuff(
        &db,
        SCE_GS_PSMCT32,
        SCREEN_WIDTH,
        SCREEN_HEIGHT,
        SCE_GS_ZALWAYS,
        SCE_GS_PSMZ24,
        SCE_GS_CLEAR);

    init();

        // Copy VU0 microcode into MicroMem0
    copyAlignedQWords((u_long128 *)0x11000000, (u_long128 *)&myVU0microprogram,
        (((u_int)&endOfProgram - (u_int)&myVU0microprogram) + 15) >> 4);

        // Copy vertices into Mem0
    copyAlignedQWords((u_long128 *)0x11004040, (u_long128 *)&Vertices, 4);

    FlushCache(0);

    dmaGif = sceDmaGetChan(SCE_DMA_GIF);

    SETVECTOR(translation,  0.0f, 0.0f, 0.0f, 0.0f);
    SETVECTOR(rotation,     0.0f, 0.0f, 0.0f, 0.0f);

    while (1) {
        sceGsSyncV(0);              // Wait for vertical blank
        buffer = 1 - buffer;
        sceGsSwapDBuff(&db, buffer);        // Swap double buffers

            // ---------------------------------------------------------
            // Calculate the rotation and Local To Screen
            // ---------------------------------------------------------

                // Rotate the planes around the axes.

        rotation[0] += delta_rot     ; if(rotation[0] > PI) rotation[0] -= 2.0f * PI;
        rotation[1] += delta_rot/2.0f; if(rotation[1] > PI) rotation[1] -= 2.0f * PI;
        rotation[2] += delta_rot/3.0f; if(rotation[2] > PI) rotation[2] -= 2.0f * PI;

                // Create Local to World matrix
        {
            sceVu0FMATRIX work;
            sceVu0UnitMatrix(work);
            sceVu0RotMatrix(work, work, rotation);
            sceVu0TransMatrix(LocalToWorldMatrix, work, translation);
        }
                // Create the final Local->Screen matrix
        sceVu0MulMatrix((void*)0x11004000, WorldToScreenMatrix, LocalToWorldMatrix);

        spr = (u_long *)SPR;
        *(spr++) = squareGIFTag[0];
        *(spr++) = squareGIFTag[1];
        *(spr++) = polyColour[0];
        *(spr++) = polyColour[1];

            // ---------------------------------------------------------
            // Perform vertex transformations
            // ---------------------------------------------------------

        asm("vcallms 0");   // Micromem addr divided by 8
        asm("vnop");

            // Copy transformed vertices

        copyAlignedQWords((u_long128 *)spr, (u_long128 *)0x11004080, 4);

        asm("sync.l");

        sceDmaSendN(dmaGif, (u_long128*)(SPR | 0x80000000), 6);
    }

    return 0;
}

void copyAlignedQWords(u_long128 *dest, u_long128 *src, int numQWords) {
    while(numQWords--) {
        *(dest++) = *(src++);
    }
}

// ==========================================================================
// Creates a World To Screen matrix
// ==========================================================================

void create_camera(
        sceVu0FMATRIX WorldToCameraMatrix,
        sceVu0FVECTOR camera_rot,
        sceVu0FVECTOR camera_p,
        sceVu0FVECTOR camera_zd,
        sceVu0FVECTOR camera_yd) {

    sceVu0FMATRIX work;
    sceVu0FVECTOR camera_p2, camera_zd2, camera_yd2;

    sceVu0UnitMatrix(work);
    sceVu0RotMatrixX(work,work,camera_rot[0]);
    sceVu0RotMatrixY(work,work,camera_rot[1]);
    sceVu0ApplyMatrix(camera_zd2, work, camera_zd);
    sceVu0ApplyMatrix(camera_yd2, work, camera_yd);
    sceVu0ApplyMatrix(camera_p2, work, camera_p);
    sceVu0CameraMatrix(WorldToCameraMatrix, camera_p2, camera_zd2, camera_yd2);
}

// ==========================================================================
// Initialise the system and program variables
// ==========================================================================
void init(void) {

            // Camera position and orientation

    sceVu0FVECTOR camera_p  = { 0.0f,  0.0f, -400.0f, 1.0f };
    sceVu0FVECTOR camera_zd = { 0.0f,  0.0f, 1.0f, 1.0f };
    sceVu0FVECTOR camera_yd = { 0.0f, -1.0f, 0.0f, 1.0f };
    sceVu0FVECTOR camera_rot = { 0.0f, 0.0f, 0.0f, 1.0f };

        // ----------------------------------------------------------------------
        // Set up lights and cameras and other matrices
        // ----------------------------------------------------------------------


    create_camera(WorldToCameraMatrix, camera_rot, camera_p, camera_zd, camera_yd);

    sceVu0ViewScreenMatrix(
        CameraToScreenMatrix,
        512.0f,
        1.0f,
        1.0f,
        2048.0f,
        2048.0f,
        1.0f,
        16777215.0f,
        64.0f,
        65536.0f
        );

    sceVu0MulMatrix(WorldToScreenMatrix, CameraToScreenMatrix, WorldToCameraMatrix);
}
```

Example 4 – VU assembler source

```
        ; these are labels to be accessed externally
    .global Vertices
    .global myVU0microprogram
    .global endOfProgram

    .p2align 4  ; This aligns the following data to quadword alignment

Vertices:
    .float  -100, -100, 0, 1
    .float   100, -100, 0, 1
    .float  -100,  100, 0, 1
    .float   100,  100, 0, 1

    .p2align 4
    .vu     ;This is required just before beginning VU opcodes.

myVU0microprogram:
    NOP                                 IADDIU      VI01, VI00, 4
    NOP                                 IADDIU      VI02, VI00, 0
    NOP                                 IADDIU      VI03, VI00, 4
    NOP                                 IADDIU      VI04, VI00, 8

    NOP                                 LQI.xyzw    VF04, (VI02++)
    NOP                                 LQI.xyzw    VF05, (VI02++)
    NOP                                 LQI.xyzw    VF06, (VI02++)
    NOP                                 LQI.xyzw    VF07, (VI02++)

loop:
    NOP                                 LQI.xyzw    VF01, (VI03++)
    MULAx       ACC, VF04, VF01x        NOP
    MADDAy      ACC, VF05, VF01y        NOP
    MADDAz      ACC, VF06, VF01z        NOP
    MADDw       VF02, VF07, VF01w       NOP
    NOP                                 DIV Q, VF00w, VF02w
    NOP                                 WAITQ
    MULQ.xyzw   VF02, VF02, Q           NOP
    FTOI4.xyzw  VF02, VF02              NOP
    NOP                                 SQI.xyzw VF02, (VI04++)

    NOP                                 ISUBIU VI01, VI01, 1
    NOP                                 NOP
    NOP                                 IBNE VI01, VI00, loop
    NOP                                 NOP
    NOP[e]                              NOP
endOfProgram:
```

Example 5 – C source

```
#include <eekernel.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <eeregs.h>
#include <libgraph.h>
#include <libdma.h>
#include <libvu0.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 448
#define SPR 0x70000000
#define PI (3.141592f)
#define SETVECTOR(v,x,y,z,w)  ((v)[0]=x,(v)[1]=y,(v)[2]=z,(v)[3]=w)

extern sceVu0FMATRIX LocalToScreenMatrix __attribute__((section(".vudata")));
extern u_long128 MatrixAndStart __attribute__((section(".vudata")));
extern u_long128 ProgramAndData __attribute__((section(".vudata")));

u_long polyColour[2] __attribute__ ((aligned(16))) = {
    0x0000000000000080L, 0x0000000000000080L
};

const u_long squareGIFTag[2] = {
    SCE_GIF_SET_TAG(
        1,          // NLOOP (Leave at 0, libraries fill this in)
        1,          // EOP (End Of Packet)
        1,          // PRE (PRIM field Enable)
        SCE_GS_SET_PRIM(    // PRIM field if PRE = 1
            SCE_GS_PRIM_TRISTRIP,   // PRIM (Type of drawing primitive)
            0,                      // IIP (Flat Shading/Gouraud Shading)
            0,                      // TME (Texture Mapping Enabled)
            0,                      // FGE (Fogging Enabled)
            0,                      // ABE (Alpha Blending Enabled)
            0,                      // AA1 (1 Pass Anti-Aliasing Enabled)
            0,                      // FST (Method of specifying Texture Coords)
            0,                      // CTXT (Context)
            0                       // FIX (Fragment Value Control)
            ),
        0,          // FLG (Data format, 0 = PACKED, 1 = REGLIST, 2 = IMAGE)
        5),         // NREG (Number of registers)
    0x55551L
};

sceVu0FMATRIX                       // Vertex transformation matrices
    *pLocalToScreenMatrix,
    LocalToWorldMatrix,
    WorldToScreenMatrix,
    CameraToScreenMatrix,
    WorldToCameraMatrix;

sceVu0FVECTOR rotation;             // Holds the vector that is used to create
                                    // the rotation matrix
sceVu0FVECTOR translation;          // The translation portion of the rotation matrix

void init(void);
void copyAlignedQWords(u_long128 *dest, u_long128 *src, int numQWords);

int main(void){
    int buffer = 0;                 // current display buffer
    sceGsDBuff db;                  // Double buffer handler
    sceDmaChan *dmaGif, *dmaVif0;               // GIF DMA handler
    u_long *spr;                    // Pointer to scratch pad ram
    float delta_rot = 1.0f * PI / 180.0f;       // change in rotation

    sceGsResetPath();
    sceDmaReset(1);

    sceGsResetGraph(
        0,
        SCE_GS_INTERLACE,
        SCE_GS_NTSC,
        SCE_GS_FIELD);

    sceGsSetDefDBuff(
        &db,
        SCE_GS_PSMCT32,
        SCREEN_WIDTH,
        SCREEN_HEIGHT,
        SCE_GS_ZALWAYS,
        SCE_GS_PSMZ24,
        SCE_GS_CLEAR);

    dmaGif = sceDmaGetChan(SCE_DMA_GIF);
    dmaVif0 = sceDmaGetChan(SCE_DMA_VIF0);
    dmaVif0->chcr.TTE = 1;

    init();

    FlushCache(0);

        // Transfer vertices and VU0 microcode into MicroMem0
    sceDmaSend(dmaVif0, &ProgramAndData);

    pLocalToScreenMatrix =
        (sceVu0FMATRIX *)(((u_int)&LocalToScreenMatrix & 0x0fffffff) | 0x20000000);

    SETVECTOR(translation,  0.0f, 0.0f, 0.0f, 0.0f);
    SETVECTOR(rotation,     0.0f, 0.0f, 0.0f, 0.0f);

    while (1) {
        sceGsSyncV(0);              // Wait for vertical blank
        buffer = 1 - buffer;
        sceGsSwapDBuff(&db, buffer);        // Swap double buffers

            // ---------------------------------------------------------
            // Calculate the rotation and Local To Screen
            // ---------------------------------------------------------

                // Rotate the planes around the axes.

        rotation[0] += delta_rot     ; if(rotation[0] > PI) rotation[0] -= 2.0f * PI;
        rotation[1] += delta_rot/2.0f; if(rotation[1] > PI) rotation[1] -= 2.0f * PI;
        rotation[2] += delta_rot/3.0f; if(rotation[2] > PI) rotation[2] -= 2.0f * PI;
```

```
                // Create Local to World matrix
        {
            sceVu0FMATRIX work;
            sceVu0UnitMatrix(work);
            sceVu0RotMatrix(work, work, rotation);
            sceVu0TransMatrix(LocalToWorldMatrix, work, translation);
        }

                // Create the final Local->Screen matrix
        sceVu0MulMatrix(*pLocalToScreenMatrix,
            WorldToScreenMatrix, LocalToWorldMatrix);

                // Upload matrix and activate program

        asm("sync.l");
        sceDmaSend(dmaVif0, &MatrixAndStart);

        spr = (u_long *)SPR;
        *(spr++) = squareGIFTag[0];
        *(spr++) = squareGIFTag[1];
        *(spr++) = polyColour[0];
        *(spr++) = polyColour[1];

            // Wait for DMA and program to complete
        while(dmaVif0->chcr.STR == 1);
        asm("vnop");

            // Copy transformed vertices

        copyAlignedQWords((u_long128 *)spr, (u_long128 *)0x11004080, 4);

        asm("sync.l");
        sceDmaSendN(dmaGif, (u_long128*)(SPR | 0x80000000), 6);
    }

    return 0;
}

void copyAlignedQWords(u_long128 *dest, u_long128 *src, int numQWords) {
    while(numQWords--) {
        *(dest++) = *(src++);
    }
}

// ===============================================================================
// Creates a World To Screen matrix
// ===============================================================================

void create_camera(
        sceVu0FMATRIX WorldToCameraMatrix,
        sceVu0FVECTOR camera_rot,
        sceVu0FVECTOR camera_p,
        sceVu0FVECTOR camera_zd,
        sceVu0FVECTOR camera_yd) {

    sceVu0FMATRIX work;
    sceVu0FVECTOR camera_p2, camera_zd2, camera_yd2;

    sceVu0UnitMatrix(work);
    sceVu0RotMatrixX(work,work,camera_rot[0]);
    sceVu0RotMatrixY(work,work,camera_rot[1]);
    sceVu0ApplyMatrix(camera_zd2, work, camera_zd);
    sceVu0ApplyMatrix(camera_yd2, work, camera_yd);
    sceVu0ApplyMatrix(camera_p2, work, camera_p);
    sceVu0CameraMatrix(WorldToCameraMatrix, camera_p2, camera_zd2, camera_yd2);
}

// ===============================================================================
// Initialise the system and program variables
// ===============================================================================
void init(void) {

            // Camera position and orientation

    sceVu0FVECTOR camera_p  = { 0.0f,  0.0f, -400.0f, 1.0f };
    sceVu0FVECTOR camera_zd = { 0.0f,  0.0f, 1.0f, 1.0f };
    sceVu0FVECTOR camera_yd = { 0.0f, -1.0f, 0.0f, 1.0f };
    sceVu0FVECTOR camera_rot = { 0.0f, 0.0f, 0.0f, 1.0f };

        // ---------------------------------------------------------------------
        // Set up lights and cameras and other matrices
        // ---------------------------------------------------------------------
    create_camera(WorldToCameraMatrix, camera_rot, camera_p, camera_zd, camera_yd);

    sceVu0ViewScreenMatrix(
        CameraToScreenMatrix,
        512.0f,
        1.0f,
        1.0f,
        2048.0f,
        2048.0f,
        1.0f,
        16777215.0f,
        64.0f,
        65536.0f
        );
    sceVu0MulMatrix(WorldToScreenMatrix, CameraToScreenMatrix, WorldToCameraMatrix);
}
```

Example 5 – VU assembler source

```
    .global MatrixAndStart
    .global LocalToScreenMatrix
    .global ProgramAndData

    .p2align 4  ; This aligns the following data to quadword alignment

    .DmaPackVif 1   ; specifies that the DMA tag will be packed with VIFcodes

MatrixAndStart:
    DMAend *
    UNPACK 4, 4, V4_32, 0, *        ; Actually two VIF instructions
LocalToScreenMatrix:                ; // Transform matrix
    .float  0, 0, 0, 0
    .float  0, 0, 0, 0
    .float  0, 0, 0, 0
    .float  0, 0, 0, 0
    .EndUnpack
    MSCAL 0             ; // kick off program
    .EndDmaData

    .p2align 4
ProgramAndData:
    DMAend *
    UNPACK 4, 4, V4_32, 4, *    ;// Vertices
    .float  -100, -100, 0, 1
    .float   100, -100, 0, 1
    .float  -100,  100, 0, 1
    .float   100,  100, 0, 1
    .EndUnpack

    MPG 0, *        ; Upload program to address 0

    .vu     ;Must be used immediately before defining a VU microprogam
    NOP                                 IADDIU      VI01, VI00, 4
    NOP                                 IADDIU      VI02, VI00, 0
    NOP                                 IADDIU      VI03, VI00, 4
    NOP                                 IADDIU      VI04, VI00, 8
    NOP                                 LQI.xyzw    VF04, (VI02++)
    NOP                                 LQI.xyzw    VF05, (VI02++)
    NOP                                 LQI.xyzw    VF06, (VI02++)
    NOP                                 LQI.xyzw    VF07, (VI02++)
loop:
    NOP                                 LQI.xyzw    VF01, (VI03++)
    MULAx       ACC, VF04, VF01x        NOP
    MADDAy      ACC, VF05, VF01y        NOP
    MADDAz      ACC, VF06, VF01z        NOP
    MADDw       VF02, VF07, VF01w       NOP
    NOP                                 DIV Q, VF00w, VF02w
    NOP                                 WAITQ
    MULQ.xyzw   VF02, VF02, Q           NOP
    FTOI4.xyzw  VF02, VF02              NOP
    NOP                                 SQI.xyzw VF02, (VI04++)
    NOP                                 ISUBIU VI01, VI01, 1
    NOP                                 NOP
    NOP                                 IBNE VI01, VI00, loop
    NOP                                 NOP
    NOP[e]                              NOP

    .EndMPG
    .EndDmaData
```

Example 6 – C source

```
// Example 6 - VU1 microprogram that creates the GS packets and sends them via PATH 1

#include <eekernel.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <eeregs.h>
#include <libgraph.h>
#include <libdma.h>
#include <libvu0.h>
#include <malloc.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 448
#define SPR 0x70000000
#define PI (3.141592f)
#define SETVECTOR(v,x,y,z,w)  ((v)[0]=x,(v)[1]=y,(v)[2]=z,(v)[3]=w)

extern sceVu0FMATRIX LocalToScreenMatrix __attribute__((section(".vudata")));
extern u_long128 MatrixAndStart __attribute__((section(".vudata")));
extern u_long128 ProgramAndData __attribute__((section(".vudata")));

sceVu0FMATRIX                      // Vertex transformation matrices
    *pLocalToScreenMatrix,
    LocalToWorldMatrix,
    WorldToScreenMatrix,
    CameraToScreenMatrix,
    WorldToCameraMatrix;

sceVu0FVECTOR rotation;            // Holds the vector that is used to create
                                   // the rotation matrix
sceVu0FVECTOR translation;         // The translation portion of the rotation matrix

void init(void);

int main(void){
    int buffer = 0;                    // current display buffer
    sceGsDBuff db;                 // Double buffer handler
    sceDmaChan *dmaVif1;               // GIF DMA handler
    float delta_rot = 1.0f * PI / 180.0f;       // change in rotation

    sceGsResetPath();
    sceDmaReset(1);

    sceGsResetGraph(
        0,
        SCE_GS_INTERLACE,
        SCE_GS_NTSC,
        SCE_GS_FIELD);

    sceGsSetDefDBuff(
        &db,
        SCE_GS_PSMCT32,
        SCREEN_WIDTH,
        SCREEN_HEIGHT,
        SCE_GS_ZALWAYS,
        SCE_GS_PSMZ24,
        SCE_GS_CLEAR);

    dmaVif1 = sceDmaGetChan(SCE_DMA_VIF1);
    dmaVif1->chcr.TTE = 1;
    init();

    FlushCache(0);

        // Transfer vertices and VU0 microcode into MicroMem0
    sceDmaSend(dmaVif1, &ProgramAndData);

    pLocalToScreenMatrix = (sceVu0FMATRIX *)(((u_int)&LocalToScreenMatrix &
0x0fffffff) | 0x20000000);

    SETVECTOR(translation,  0.0f, 0.0f, 0.0f, 0.0f);
    SETVECTOR(rotation,     0.0f, 0.0f, 0.0f, 0.0f);

    while (1) {
        sceGsSyncV(0);              // Wait for vertical blank
        buffer = 1-buffer;
        sceGsSwapDBuff(&db, buffer);        // Swap double buffers

            // ------------------------------------------------------------
            // Calculate the rotation and Local To Screen
            // ------------------------------------------------------------

                // Rotate the planes around the axes.

        rotation[0] += delta_rot     ; if(rotation[0] > PI) rotation[0] -= 2.0f * PI;
        rotation[1] += delta_rot/2.0f; if(rotation[1] > PI) rotation[1] -= 2.0f * PI;
        rotation[2] += delta_rot/3.0f; if(rotation[2] > PI) rotation[2] -= 2.0f * PI;

                // Create Local to World matrix
        {
            sceVu0FMATRIX work;
            sceVu0UnitMatrix(work);
            sceVu0RotMatrix(work, work, rotation);
            sceVu0TransMatrix(LocalToWorldMatrix, work, translation);
        }

                // Create the final Local->Screen matrix
        sceVu0MulMatrix(*pLocalToScreenMatrix,
             WorldToScreenMatrix, LocalToWorldMatrix);

                // Upload matrix and activate program

        asm("sync.l");
        sceDmaSend(dmaVif1, &MatrixAndStart);
    }

    return 0;
}


// ============================================================================
// Creates a World To Screen matrix
// ============================================================================

void create_camera(
        sceVu0FMATRIX WorldToCameraMatrix,
        sceVu0FVECTOR camera_rot,
        sceVu0FVECTOR camera_p,
        sceVu0FVECTOR camera_zd,
        sceVu0FVECTOR camera_yd) {

    sceVu0FMATRIX work;
    sceVu0FVECTOR camera_p2, camera_zd2, camera_yd2;

    sceVu0UnitMatrix(work);
    sceVu0RotMatrixX(work,work,camera_rot[0]);
    sceVu0RotMatrixY(work,work,camera_rot[1]);
    sceVu0ApplyMatrix(camera_zd2, work, camera_zd);
    sceVu0ApplyMatrix(camera_yd2, work, camera_yd);
    sceVu0ApplyMatrix(camera_p2, work, camera_p);
    sceVu0CameraMatrix(WorldToCameraMatrix, camera_p2, camera_zd2, camera_yd2);
}

// ============================================================================
// Initialise the system and program variables
// ============================================================================
void init(void) {
        // Camera position and orientation

    sceVu0FVECTOR camera_p  = { 0.0f,  0.0f, -400.0f, 1.0f };
    sceVu0FVECTOR camera_zd = { 0.0f,  0.0f, 1.0f, 1.0f };
    sceVu0FVECTOR camera_yd = { 0.0f, -1.0f, 0.0f, 1.0f };
    sceVu0FVECTOR camera_rot = { 0.0f, 0.0f, 0.0f, 1.0f };

        // -------------------------------------------------------------------------
        // Set up lights and cameras and other matrices
        // -------------------------------------------------------------------------


    create_camera(WorldToCameraMatrix, camera_rot, camera_p, camera_zd, camera_yd);

    sceVu0ViewScreenMatrix(
        CameraToScreenMatrix,
        512.0f,
        1.0f,
        1.0f,
        2048.0f,
        2048.0f,
        1.0f,
        16777215.0f,
        64.0f,
        65536.0f
        );

    sceVu0MulMatrix(WorldToScreenMatrix, CameraToScreenMatrix, WorldToCameraMatrix);
}
```

Example 6 – VU assembler source

```
        ; these are labels we will want to access externally
    .global MatrixAndStart
    .global LocalToScreenMatrix
    .global ProgramAndData

    .p2align 4  ; This aligns the following data to quadword alignment

    .DmaPackVif 1   ; specifies that the DMA tag will be packed with VIFcodes

MatrixAndStart:
    DMAend *

    UNPACK 4, 4, V4_32, 0, *        ; Actually two VIF instructions
LocalToScreenMatrix:                ; // Transform matrix
    .float  0, 0, 0, 0
    .float  0, 0, 0, 0
    .float  0, 0, 0, 0
    .float  0, 0, 0, 0
    .EndUnpack

    MSCAL 0             ; // kick off program
    .EndDmaData

;   .p2align 4
ProgramAndData:

    DMAend *

    UNPACK 4, 4, V4_32, 4, *    ;// Vertices
    .float  -100, -100, 0, 1
    .float   100, -100, 0, 1
    .float  -100,  100, 0, 1
    .float   100,  100, 0, 1

    GIFpacked PRIM=0X04, REGS={ RGBAQ, XYZ2, XYZ2, XYZ2, XYZ2 }, NLOOP=1, EOP
    .EndGif
    .word   0x80, 0x80, 0, 0    ; Colour 1
    .EndUnpack


    MPG 0, *        ; Upload program to address 0

    .vu     ;Must be used immediately before defining a VU microprogam
    NOP                                 IADDIU      VI01, VI00, 4
    NOP                                 IADDIU      VI02, VI00, 0
    NOP                                 IADDIU      VI03, VI00, 4
    NOP                                 IADDIU      VI04, VI00, 10

    NOP                                 LQI.xyzw    VF04, (VI02++)
    NOP                                 LQI.xyzw    VF05, (VI02++)
    NOP                                 LQI.xyzw    VF06, (VI02++)
    NOP                                 LQI.xyzw    VF07, (VI02++)

loop:
    NOP                                 LQI.xyzw    VF01, (VI03++)
    MULAx       ACC, VF04, VF01x        NOP
    MADDAy      ACC, VF05, VF01y        NOP
    MADDAz      ACC, VF06, VF01z        NOP
    MADDw       VF02, VF07, VF01w       NOP
    NOP                                 DIV Q, VF00w, VF02w
    NOP                                 WAITQ
    MULQ.xyzw   VF02, VF02, Q           NOP
    FTOI4.xyzw  VF02, VF02              NOP
    NOP                                 SQI.xyzw VF02, (VI04++)

    NOP                                 ISUBIU VI01, VI01, 1
    NOP                                 NOP
    NOP                                 IBNE VI01, VI00, loop
    NOP                                 NOP
    NOP                                 IADDIU      VI04, VI00, 8   ; Addr of GIFtag
    NOP[e]                              XGKICK      VI04

    .EndMPG
    .EndDmaData
```

Example 7 – C source

```
#include <eekernel.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <eeregs.h>
#include <libgraph.h>
#include <libdma.h>
#include <libvu0.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 224       // NOTE: This is now half the size of the original
#define SPR 0x70000000
#define PI (3.141592f)
#define SETVECTOR(v,x,y,z,w)  ((v)[0]=x,(v)[1]=y,(v)[2]=z,(v)[3]=w)

extern sceVu0FMATRIX LocalToScreenMatrix __attribute__((section(".vudata")));
extern u_long128 MatrixAndStart __attribute__((section(".vudata")));
extern u_long128 ProgramAndData __attribute__((section(".vudata")));

sceVu0FMATRIX                       // Vertex transformation matrices
    *pLocalToScreenMatrix,
    LocalToWorldMatrix,
    WorldToScreenMatrix,
    CameraToScreenMatrix,
    WorldToCameraMatrix;

sceVu0FVECTOR rotation;             // Holds the vector that is used to create
                                    // the rotation matrix
sceVu0FVECTOR translation;          // The translation portion of the rotation matrix

void init(void);

int main(void){
    int field;                      // current display field
    sceGsDBuff db, *p_db;                   // Double buffer handler
    sceDmaChan *dmaVif1;                // GIF DMA handler
    float delta_rot = 1.0f * PI / 180.0f;       // change in rotation

    sceGsResetPath();
    sceDmaReset(1);

    sceGsResetGraph(
        0,
        SCE_GS_INTERLACE,
        SCE_GS_NTSC,
        SCE_GS_FRAME);

    sceGsSetDefDBuff(
        &db,
        SCE_GS_PSMCT32,
        SCREEN_WIDTH,
        SCREEN_HEIGHT,
        SCE_GS_ZALWAYS,
        SCE_GS_PSMZ24,
        SCE_GS_CLEAR);

    dmaVif1 = sceDmaGetChan(SCE_DMA_VIF1);
    dmaVif1->chcr.TTE = 1;

    init();

    p_db = (sceGsDBuff *)(((u_int)&db & 0x0fffffff) | 0x20000000);

    FlushCache(0);

        // Transfer vertices and VU0 microcode into MicroMem0
    sceDmaSend(dmaVif1, &ProgramAndData);

    pLocalToScreenMatrix = (sceVu0FMATRIX *)(((u_int)&LocalToScreenMatrix &
0x0fffffff) | 0x20000000);

    SETVECTOR(translation,  0.0f, 0.0f, 0.0f, 0.0f);
    SETVECTOR(rotation,     0.0f, 0.0f, 0.0f, 0.0f);

    while (1) {
        field = sceGsSyncV(0);              // Wait for vertical blank
        sceGsSwapDBuff(p_db, !field);       // Swap double buffers

            // add half pixel to Y offset address for interlace
        sceGsSetHalfOffset((field&1)?(&p_db->draw0):(&p_db->draw1),
            2048, 2048, 1-field);

            // ---------------------------------------------------------
            // Calculate the rotation and Local To Screen
            // ---------------------------------------------------------

                // Rotate the planes around the axes.

        rotation[0] += delta_rot     ; if(rotation[0] > PI) rotation[0] -= 2.0f * PI;
        rotation[1] += delta_rot/2.0f; if(rotation[1] > PI) rotation[1] -= 2.0f * PI;
        rotation[2] += delta_rot/3.0f; if(rotation[2] > PI) rotation[2] -= 2.0f * PI;

                // Create Local to World matrix
        {
            sceVu0FMATRIX work;
            sceVu0UnitMatrix(work);
            sceVu0RotMatrix(work, work, rotation);
            sceVu0TransMatrix(LocalToWorldMatrix, work, translation);
        }

                // Create the final Local->Screen matrix
        sceVu0MulMatrix(*pLocalToScreenMatrix, WorldToScreenMatrix,
                            LocalToWorldMatrix);

                // Upload matrix and activate program

        asm("sync.l");
        sceDmaSend(dmaVif1, &MatrixAndStart);
    }

    return 0;
}


// ============================================================================
// Creates a World To Screen matrix
// ============================================================================

void create_camera(
        sceVu0FMATRIX WorldToCameraMatrix,
        sceVu0FVECTOR camera_rot,
        sceVu0FVECTOR camera_p,
        sceVu0FVECTOR camera_zd,
        sceVu0FVECTOR camera_yd) {

    sceVu0FMATRIX work;
    sceVu0FVECTOR camera_p2, camera_zd2, camera_yd2;

    sceVu0UnitMatrix(work);
    sceVu0RotMatrixX(work,work,camera_rot[0]);
    sceVu0RotMatrixY(work,work,camera_rot[1]);
    sceVu0ApplyMatrix(camera_zd2, work, camera_zd);
    sceVu0ApplyMatrix(camera_yd2, work, camera_yd);
    sceVu0ApplyMatrix(camera_p2, work, camera_p);
    sceVu0CameraMatrix(WorldToCameraMatrix, camera_p2, camera_zd2, camera_yd2);
}

// ============================================================================
// Initialise the system and program variables
// ============================================================================
void init(void) {

            // Camera position and orientation

    sceVu0FVECTOR camera_p  = { 0.0f,  0.0f, -400.0f, 1.0f };
    sceVu0FVECTOR camera_zd = { 0.0f,  0.0f, 1.0f, 1.0f };
    sceVu0FVECTOR camera_yd = { 0.0f, -1.0f, 0.0f, 1.0f };
    sceVu0FVECTOR camera_rot = { 0.0f, 0.0f, 0.0f, 1.0f };

        // ----------------------------------------------------------------------
        // Set up lights and cameras and other matrices
        // ----------------------------------------------------------------------


    create_camera(WorldToCameraMatrix, camera_rot, camera_p, camera_zd, camera_yd);

    sceVu0ViewScreenMatrix(
        CameraToScreenMatrix,
        512.0f,
        1.0f,
        0.5f,          // Y scale factor
        2048.0f,
        2048.0f,
        1.0f,
        16777215.0f,
        64.0f,
        65536.0f
        );

    sceVu0MulMatrix(WorldToScreenMatrix, CameraToScreenMatrix, WorldToCameraMatrix);
}
```

Example 7 – VU assembler source

There were no changes made to the VU assembler source between example 6 and example 7.

Example 8 – C source

```
#include <eekernel.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <eeregs.h>
#include <libgraph.h>
#include <libdma.h>
#include <libvu0.h>
#include <malloc.h>
#include <sifdev.h>
#include <sifrpc.h>
#include <libpad.h>
```

```
#include <libdev.h>

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 224
#define SPR 0x70000000
#define PI (3.141592f)
#define SETVECTOR(v,x,y,z,w) ((v)[0]=x,(v)[1]=y,(v)[2]=z,(v)[3]=w)

extern sceVu0FMATRIX LocalToScreenMatrix __attribute__((section(".vudata")));
extern u_long128 MatrixAndStart __attribute__((section(".vudata")));
extern u_long128 ProgramAndData __attribute__((section(".vudata")));

sceVu0FMATRIX                       // Vertex transformation matrices
    *pLocalToScreenMatrix,
    LocalToWorldMatrix,
    WorldToScreenMatrix,
    CameraToScreenMatrix,
    WorldToCameraMatrix;

sceVu0FVECTOR rotation, translation;


    // Pad buffer
u_long128 padDma[scePadDmaBufferMax] __attribute__((aligned (64)));
u_int paddata;
u_char rdata[32];

void init(void);

int main(void){
    int field;                      // current display field
    sceGsDBuff db, *p_db;                   // Double buffer handler
    sceDmaChan *dmaVif1;                // GIF DMA handler
    float delta_rot = 1.0f * PI / 180.0f;       // change in rotation
    int console;                        // screen output debugging console identifier

    sceSifInitRpc(0);
    while(sceSifLoadModule("host0:/usr/local/sce/iop/modules/sio2man.irx",0,NULL)<0){
        printf("Retrying module sio2man\n");
    }
    while(sceSifLoadModule("host0:/usr/local/sce/iop/modules/padman.irx",0,NULL)<0){
        printf("Retrying module padman\n");
    }

    sceGsResetPath();
    sceDmaReset(1);

    scePadInit(0);                      // Open controller
    scePadPortOpen(0, 0, padDma);

    sceGsResetGraph(
        0,
        SCE_GS_INTERLACE,
        SCE_GS_NTSC,
        SCE_GS_FRAME);

    sceGsSetDefDBuff(
        &db,
        SCE_GS_PSMCT32,
        SCREEN_WIDTH,
        SCREEN_HEIGHT,
        SCE_GS_ZALWAYS,
        SCE_GS_PSMZ24,
        SCE_GS_CLEAR);

    dmaVif1 = sceDmaGetChan(SCE_DMA_VIF1);
    dmaVif1->chcr.TTE = 1;

    init();

    sceDevConsInit();   // Initialise screen console output
    console = sceDevConsOpen(
      (2048 - SCREEN_WIDTH/2 + 20) << 4,  // top left GS X primitive coord
      (2048 - SCREEN_HEIGHT/2 + 20) << 4, // top left GS Y primitive coord
      80, // Number of chars (X)
      5); // Number of chars (Y)
    sceDevConsAttribute(console, 7);    // Output in white (default colours can be
                                        // redefined by sceDevConsSetColor)

    p_db = (sceGsDBuff *)(((u_int)&db & 0x0fffffff) | 0x20000000);

    FlushCache(0);

        // Transfer vertices and VU0 microcode into MicroMem0
    sceDmaSend(dmaVif1, &ProgramAndData);

    pLocalToScreenMatrix = (sceVu0FMATRIX *)(((u_int)&LocalToScreenMatrix &
0x0fffffff) | 0x20000000);

    SETVECTOR(translation,  0.0f, 0.0f, 0.0f, 0.0f);
    SETVECTOR(rotation,     0.0f, 0.0f, 0.0f, 0.0f);

    while (1) {
        field = sceGsSyncV(0);              // Wait for vertical blank
        sceGsSwapDBuff(p_db, !field);       // Swap double buffers

            // add half pixel to Y offset address for interlace
        sceGsSetHalfOffset((field&1)?(&p_db->draw0):(&p_db->draw1),
            2048, 2048, 1-field);

            // ---------------------------------------------------------
            // Calculate the rotation and Local To Screen
            // ---------------------------------------------------------

                // Read controller
        if(scePadRead(0, 0, rdata) > 0){
            paddata = 0xffff ^ ((rdata[2] << 8) | rdata[3]);
        } else {
            paddata = 0;
        }

                // Rotate the planes around the axes.

        if(paddata & SCE_PADLup) {
            rotation[0] += delta_rot;
        } else if(paddata & SCE_PADLdown) {
            rotation[0] -= delta_rot;
        }

        if(paddata & SCE_PADLleft) {
            rotation[1] += delta_rot;
        } else if(paddata & SCE_PADLright) {
            rotation[1] -= delta_rot;
        }

        if(paddata & SCE_PADRleft) {
            rotation[2] += delta_rot;
        } else if(paddata & SCE_PADRright) {
            rotation[2] -= delta_rot;
        }

        {   // Clamp Euler rotation vector
            int i;
            for(i = 0; i < 3; i++) {
                if(rotation[i] < -PI) {
                    rotation[i] += 2.0f * PI;
                } else if(rotation[i] > PI) {
                    rotation[i] -= 2.0f * PI;
                }
            }
        }

        sceDevConsClear(console);     // Output rotation values to screen
        sceDevConsPrintf(console,
            "X rotation : %2.4f  Y rotation : %2.4f  Z rotation : %2.4f",
            rotation[0], rotation[1], rotation[2]);
        sceDevConsDraw(console);

                // Create Local to World matrix
        {
            sceVu0FMATRIX work;
            sceVu0UnitMatrix(work);
            sceVu0RotMatrix(work, work, rotation);
            sceVu0TransMatrix(LocalToWorldMatrix, work, translation);
        }

                // Create the final Local->Screen matrix
        sceVu0MulMatrix(*pLocalToScreenMatrix, WorldToScreenMatrix,
LocalToWorldMatrix);

                // Upload matrix and activate program

        asm("sync.l");
        sceDmaSend(dmaVif1, &MatrixAndStart);
    }

    return 0;
}

// ==============================================================================
// Creates a World To Screen matrix
// ==============================================================================

void create_camera(
        sceVu0FMATRIX WorldToCameraMatrix,
        sceVu0FVECTOR camera_rot,
        sceVu0FVECTOR camera_p,
        sceVu0FVECTOR camera_zd,
        sceVu0FVECTOR camera_yd) {

    sceVu0FMATRIX work;
    sceVu0FVECTOR camera_p2, camera_zd2, camera_yd2;

    sceVu0UnitMatrix(work);
    sceVu0RotMatrixX(work,work,camera_rot[0]);
    sceVu0RotMatrixY(work,work,camera_rot[1]);
    sceVu0ApplyMatrix(camera_zd2, work, camera_zd);
    sceVu0ApplyMatrix(camera_yd2, work, camera_yd);
    sceVu0ApplyMatrix(camera_p2, work, camera_p);
    sceVu0CameraMatrix(WorldToCameraMatrix, camera_p2, camera_zd2, camera_yd2);
}

// ==============================================================================
// Initialise the system and program variables
// ==============================================================================
void init(void) {

            // Camera position and orientation

    sceVu0FVECTOR camera_p  = { 0.0f,  0.0f, -400.0f, 1.0f };
    sceVu0FVECTOR camera_zd = { 0.0f,  0.0f, 1.0f, 1.0f };
    sceVu0FVECTOR camera_yd = { 0.0f, -1.0f, 0.0f, 1.0f };
    sceVu0FVECTOR camera_rot = { 0.0f, 0.0f, 0.0f, 1.0f };

        // --------------------------------------------------------------------------
        // Set up lights and cameras and other matrices
        // --------------------------------------------------------------------------

    create_camera(WorldToCameraMatrix, camera_rot, camera_p, camera_zd, camera_yd);

    sceVu0ViewScreenMatrix(
        CameraToScreenMatrix,
        512.0f,
        1.0f,
        0.5f,
        2048.0f,
        2048.0f,
        1.0f,
        16777215.0f,
        64.0f,
        65536.0f
        );

    sceVu0MulMatrix(WorldToScreenMatrix, CameraToScreenMatrix, WorldToCameraMatrix);
}
```